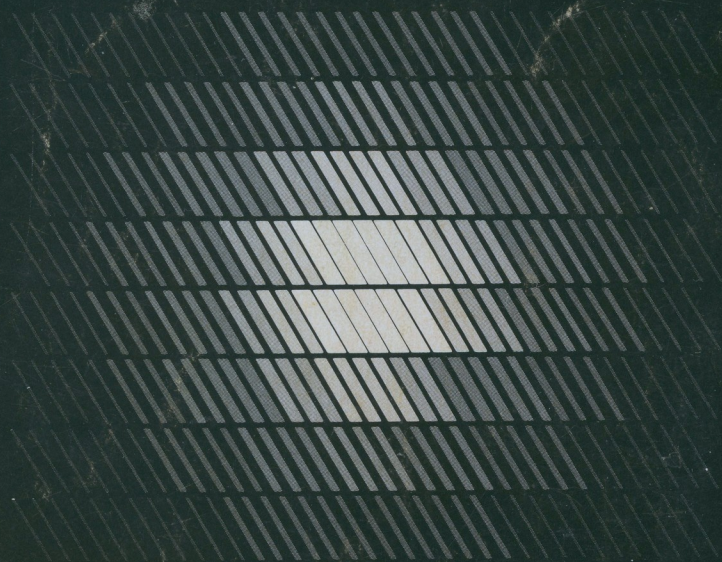


 **VideoBrain**TM

User's Manual

APL/S
The Computational Language



APL/S The Computational Language

User's Manual

An Introduction
to APL/S, A Programming Language
for the VideoBrain Home Computer

VideoBrain Computer Company
2950 Patrick Henry Drive
Santa Clara, CA 95050
(408) 988-3020

Wallace Judd
Simon Cintz
Education Technology Interface
378 Cambridge Avenue
Palo Alto, CA 94306
(415) 493-4242

APL/S designed and implemented
by Robert G. Brown.
777 S. Mathilda Ave.
Suite D148
Sunnyvale, CA 94087
(408) 737-1224

All rights reserved including the right of
reproduction in whole or in part in any form.

© 1978 VideoBrain Computer Co.
P/N 741-0026 Rev. 0 11/78

Price: \$15.00

Printed in the U.S.A.

Preface

This manual gives a step-by-step introduction to APL/S, the powerful programming language of VideoBrain. In it are many clear, simple programs ready to key into your VideoBrain, with commentary. In a few hours of work with your VideoBrain you, too, will have **PROGRAMMING POWER**.

Table of Contents

- 1. Getting Started 1
 - 1 Introduction 1
 - 2 Inserting the Cartridge 2
 - 3 The Special Keys 2
 - 4 Simple Numerical Calculations 3
 - 5 Row Numbers 4
 - 6 BARHeight and BARColor 4
 - 7 BARColor Chart 5
 - 8 What Is a Line? 5
 - 9 Assigning Values to Variables 6
 - 10 Using Variables 7
 - 11 Precedence 8
 - 12 Row Variables 9
 - 13 Computation with Row Variables 9
 - 14 Input/Output 11
- 2. Beginning Programming 13
 - 1 Your First Computer Program 13
 - 2 Making Changes in Your Program 15
 - 3 Edit Command Summary 16
 - 4 Counters and Assignment 17
 - 5 The KEYBoard and " " Commands 18
 - 6 Using Prompts in Programs 19
 - 7 A Program of Your Very Own 20
 - 8 Three Kinds of Words 21
- 3. Flow of Control 23
 - 1 Blocks 23
 - 2 IF . . THEN . . ELSE . . ENDIf 23
 - 3 Conditionals 24
 - 4 Example 25
 - 5 Bigger Blocks 25
 - 6 New Lines for THEN and ELSE 26
 - 7 Omitting ELSE 26
 - 8 Multiple Conditions and NOT 26
 - 9 WHILE . . DO . . ENDWhile 27
 - 10 EXIT Command 29
 - 11 Nested Structures 30

4. Row Variables	.31
-1 Operations with Two Row Variables	.32
-2 Reduction Operator	.33
-3 Using Individual Values of a Row Variable	.34
-4 The Join Function	.35
-5 The SIZE Function	.35
-6 The MINimum and MAXimum Functions	.36
-7 Some Technical Terms	.37
-8 The ARRAY Function	.37
-9 The INDEX Function	.38
-10 Reduction with Logical Expressions	.38
-11 MAX and MIN with 2 Arrays	.39
-12 The Comma as Ravel	.39
-13 Programming Practice	.40
5. Saving and Storing Programs	.43
-1 The Concept of Workspace	.43
-2 SAVE	.43
-3 LOAD	.45
-4 VERIfy	.46
-5 LOAD and SAVE Example	.47
6. A Library of Sample Programs	.49
-1 The Piracy Approach to Programming	.49
-2 Using a Utility Program	.49
-3 Designing Utility Programs	.50
-4 Utility Programs	.51
-5 SCALE	.52
-6 Inverse Trig Functions	.53
-7 COLOR	.55
-8 Amusing Programs	.57
7. Debugging	.61
-1 Avoiding Bugs in Your Programs	.61
-2 Four Types of Error Messages	.61
-3 Immediate Mode Errors	.62
-4 Edit Mode Errors	.63
-5 Error Messages Upon Leaving Edit Mode	.63
-6 Run Time Errors	.64
-7 Infrequent Errors	.66
-8 Infinite Loops	.67
-9 Conceptual Errors	.68
-10 ONTRace, OFFTrace and <STOP>	.69

8. Alphabetical Reference Section 71

 -1 Commands 71

 -2 Statements 75

 -3 Functions 86

Appendices 125

 A Syntax Diagrams 125

 B Technical Notes 131

 C Error Messages 133

 D Topic Index 137

 E Application Program Index

 F Glossary of Terms 141

 G Storage Management 143

 H Reserved Words 145

Chapter 1: Getting Started

-1 Introduction

Congratulations on deciding to become part of the computer age — with your purchase of a VideoBrain home computer. This instruction manual will be your guide to the computer and its language, starting from turning the machine on all the way to sophisticated use of the powerful features of the language.

Using the APL/S program cartridge, your VideoBrain doesn't take commands in English — it only understands commands in APL/S, its language.

The best method for using this manual is to connect your VideoBrain to your television set (directions are in your VideoBrain Owner's Manual) and go through the examples one by one. Type each program into your computer, run it, and try to modify it. Some of the examples may seem trivial or silly — others may have to be reviewed several times before they are clear. But by the time you have worked through a number of them, the APL/S language will be yours. Soon you will be able to make up your own programs to do budgeting, accounting, or any number of useful jobs around your home or business.

If you have done some programming in another language, you may want to look at Chapter 8, which is the Reference Section. The program library in Chapter 6 will also give you a good idea of the structure and capabilities of the language.

If this is your first experience programming, or you don't feel confident in your first language, this book is for you. Relax and start using the keys as soon as you're ready to go. Remember that there is no series of keys you can use on the keyboard which will do any damage to the VideoBrain itself. You might erase a few of your own programs from computer memory — but there is nothing which will hurt the machine itself.

Most lessons in this manual are presented as problems. You will be given the problem, and a short example of how to think the problem through. Then a sample program will be given for you to key into your VideoBrain. The sample program solves the basic problem. Then you will be asked to modify the sample program in some way. Finally, some problems end with more sophisticated tricks, a sort of 'getting fancy' section for hotshots.

Feel free to experiment. Play with commands you learn. If a new word is used in a description, look it up in the glossary, Appendix F. The glossary gives examples and explains each word in simple language. Use it often. For further explanations and examples, refer to the reference section in Chapter 8.

Chapter 1: Getting Started

As a result of going through this book, you will emerge with two things:

- 1) You will know how to think through a problem, analyze it into steps, and translate those steps into the APL/S programming language.
- 2) In addition, you will have a small but powerful library of programs which you understand. These programs will become the core of a more extensive library of many more programs you will want to develop to exploit the potential of VideoBrain for your use.

-2 Inserting the Cartridge

1. Make sure your VideoBrain computer is attached to your TV as described in the Owner's Manual. Check that power is on.
2. Push the cartridge carrier release button above the VideoBrain keyboard in order to swing the cartridge carrier door up.
3. With the label facing up, slide the cartridge all the way into the tracks suspended from the cartridge carrier door.
4. Gently push the cartridge door down into the computer until it locks.
5. Push the *Master Control* button. The title of the cartridge should appear on your TV.

-3 The Special Keys

Now that your VideoBrain is ready to go, you should first get used to the blue keys in the lower part of the keyboard. In this manual, we will name the keys by the words printed on the lower half, since words on the upper part refer to things VideoBrain does when APL/S is not being used.

⟨SPACE⟩ moves the cursor over a space each time you use it. If you go past the end of a line, another line is started. The next line is marked with a white square at the beginning, to show that it continues the line above.

⟨BACK⟩ means BACKspace. You can use it to back the cursor up. The ⟨BACK⟩ key erases letters that it backs up over. If you type in a line that is more than fifteen characters long, you can backspace up onto the previous line.

⟨NEXT⟩ moves you to the next logical line. The ⟨NEXT⟩ key has to be used after each logical line to enter the line into the computer.

⟨SHIFT⟩ lets you put in the upper or lower symbols on the black keys. Notice that when you press the ⟨SHIFT⟩ key, the cursor changes from "L" (meaning lower case) to "U" (for upper case). The cursor will stay at whatever case it is in until you press the ⟨SHIFT⟩ key again.

⟨ERASE⟩ erases the entire line. It puts a white box at the end of the line, and moves the cursor on to the next line.

⟨SPECIAL⟩ and ⟨PREVIOUS⟩ are not important until we do some programming in edit mode in Chapter 2.

⟨MASTER CONTROL⟩ is like the CLEAR key on a calculator. It erases everything, and begins with a totally clear screen. Any time you have trouble, you can press ⟨MASTER CONTROL⟩ and start all over again.

–4 Simple Numerical Calculations

Right now, the computer is in immediate mode. That means that it will immediately do whatever you tell it to (using APL/S, and with good grammar, of course). Any simple calculation that you want done, it can do without having to be programmed.

Press ⟨MASTER CONTROL⟩, and make sure that the cursor shows that you are in upper case. If you are not in upper case, press the ⟨SHIFT⟩ key. Type the following lines, using ⟨NEXT⟩ after each one:

54 + 23	The answer is 77.
54 + 23 =	Notice = is not right, here.

With the second statement, you get a message that reads SYNTAX 6. All the error messages noted as SYNTAX or VALUE are listed in the back of this manual, in Appendix C. Usually when you get a SYNTAX error message, VideoBrain gives you the line so you can correct it and try it again. In this case, just key ⟨BACK⟩ to back up one space and erase the equals sign. Then press ⟨NEXT⟩ to enter the line into the computer again. This time you get the correct answer, 77.

Try some of the following computations — again making sure that you're in upper case. **Press ⟨NEXT⟩ after each line in this example, and in all the examples in this manual.**

2 + 5	Did you key ⟨NEXT⟩ after the line?
23 x 10	Note that "x" is not the letter "X" but rather the times symbol.

44 ÷ 4	
1111+2222+3333+4444	Even though the line gets broken, VideoBrain computes as though it were all on one line.

Play with the calculations, trying different numbers. Experiment with parentheses (), the decimal point, the minus key, and if you're really daring, with the cents symbol (¢) (which is not cents, but the negative sign).

-5 Row Numbers

So far, you have used the computer like a very expensive calculator. With the next few examples you will begin to see how much more powerful VideoBrain is than a simple calculator. Make sure that you leave spaces in the text between the numbers. The spaces in the text between numbers are very important.

1 2 3 + 4	Did you remember <NEXT> after the line?
1 2 3 x 10	Again, not "X" but times.
10 20 30 ÷ 10	This should give 10,20,30
9 8 7 6 - 5	

The things above are called row numbers or one dimensional arrays. APL/S lets you treat all the numbers separated by spaces as though they were a single number. Here's how it works:

You Type	What VideoBrain Does
1 2 3 + 4	$\begin{array}{r} 1 \quad 2 \quad 3 \\ +4 \quad +4 \quad +4 \\ \hline 5 \quad 6 \quad 7 \end{array}$

Notice that APL/S performs the same operation on all the numbers in the row. You can experiment with row numbers in lots of ways. Below are some examples which will get you started in your experiments.

4 5 6 - 1	Subtracts 1 from each.
2 3 4 x 10	Ten times each number.
10 x 2 3 4	Order doesn't matter!
100 + 1 2 3 4	

After you've experimented some, try to answer some of the following questions, using APL/s.

- 1) Can you make a list of the 4 times tables, from 1 to 10?
- 2) With one statement, make a list of the 13 times tables, from 1 to 13.
- 3) You wanted to discount all prices by 20%. The items used to be 2.98, 3.56, and 11.41. What is the new price of each item?

-6 BARHeight and BARColor

So far, you have only needed to input upper case characters -- numbers and symbols. For the examples in this section, you'll need to use both upper and lower case characters, so remember to use the <SHIFT> key when changing from letters to numbers, and back again.

The top half of your APL/S screen is there to give you graphic display of your answers. In the following example as in all the examples in this book, type in only the bold, capital letters and the numbers that follow them. The other letters are only there to help you remember what the short words stand for. VideoBrain will only remember the first four letters you type, no matter how many you use.

BARHeight 48	Did you press <NEXT>?
BARColor 9	Changes the bar color to violet.
BARHeight 10 20 30	Makes 3 bars.
BARColor 4 14 1	Changes colors to red yellow blue.
BARHeight 40 50 60 70 80 90 100	Maximum height is 64.
BARColor 9 9 9 9 9 9 9	Turns all the bars violet.

Experiment with the bars. The color numbers go from 0 to 15, though you can use larger numbers. The maximum and minimum heights of the bars are +64 and -64. There are seven bars, and they can be used for all sorts of interesting displays. If you want to see some unique uses, try some of the amusing programs in Chapter 6.

-7 BARColor Chart

The colors of the bars are included in the chart below. You can experiment with them as you do further examples with the **BARColor** command.

BARColor Command Codes

0 Black	8 Grey
1 Blue	9 Violet
2 Green	10 Light Green
3 Aqua	11 Light Blue
4 Red	12 Orange
5 Purple	13 Light Purple
6 Brown	14 Yellow
7 Silver	15 White

If you need to change a line, just key <BACK> until you're under the mistake, then type the line from there on over again. The cursor always wipes out letters as it backspaces. Don't forget that you always have to key <NEXT> to tell the computer to take in a line.

-8 What Is a Line?

While it may seem obvious what a line is to you, to the computer it is not so clear. The computer has two kinds of lines, a visual line and a logical line.

A visual line is sixteen characters long. A logical line ends whenever the key <NEXT> is pushed. The maximum logical line is 125 characters long, although you won't often need to make a line that long. Logical lines and visual lines are very different. Key in the following example to see just how different.

BARHeight 64 43 11 14 23 48<NEXT>	One logical line.
BARHeight 0 0 0 0 0 0<NEXT>	Clears bars.
BARHeight 64 43 11 14<NEXT>23 48<NEXT>	Two logical lines.
TOTAL = 1+20+300+4000+500000<NEXT>	One logical line.
TOTAL<NEXT>	
TOTAL = 1+200+300<NEXT>+4000+50000<NEXT>	Two logical lines.
TOTAL<NEXT>	

Chapter 1: Getting Started

Notice that the values beyond BARHeight are not added together — they are just separated by a space.

As long as you have not pushed the <NEXT> key, VideoBrain takes in everything you have typed as though it were one, long line. This is the logical line. The 'picture' of the line may be broken up into two or three lines, but the computer sees all the characters up to <NEXT> as a single line.

When you type <ERASE>, it erases the logical line, not the visual line. Even though you still see the visual line on the screen, it has been erased from computer memory.

In this manual, it is assumed that you type <NEXT> after each line of program in the examples.

-9 Assigning Values to Variables

In APL/S, as in other computer languages, there are things called variables. A variable is a word that you, the user, make up. The word is given (or assigned) a value, using the equal sign. When the equal sign appears to the right of a user-made word, the word is given the value which appears to the right of the equals sign.

42 x 10	Key <NEXT> after each line.
15 + 6	
3.50 - 1.85	
COST = 48	This gives the value 48 to COST.
COST	Show the value of COST.
COST + 8	Show the value of COST + 8.

Notice that after the line **COST** = 48, the computer did not show a number. This is because in immediate mode, the computer only prints a value after a line where there is no = sign. The = sign assigns a value to the name of the variable on the left. It only works one way. Look at the statements below. You can try them in your computer if you'd like. Type the wrong way first, and see the types of error statements you get when you do it wrong. If you use <MASTER CONTROL> to get out of any problem situations, remember that all the variables are erased from the computer memory.

WRONG	RIGHT	REASON
48 = COST	COST = 48	The variable has to be left of =.
COST = \$32.50	COST = 32.50	You can not use \$ in statements.
TYPE COST	COST	Naming the variable shows its value.
48 = 48	NMBR = 48	A number cannot show to the left of =.
24 = 10+14	TOTL = 10+14	A number cannot show to the left of =.

Now, you try some things. Print the value of **TOTL**. Give a variable named **PRC** the value 16.00. Print the value of **PRC**. Give a variable named **DISC** the value 16.00 x .05. Print **DISC**.

The words **COST**, **TOTL**, **PRC**, and **DISC** in the statements you have been using are called variables. A variable takes whatever value is assigned to it, and a value is assigned to a variable by the = sign. Notice that the = sign is not the same as the = sign in arithmetic. Here, = tells the computer to find the value on the right of the = sign, and assign that value to the variable whose name is on the left of the = sign.

The following examples should give you a better idea of the way a variable gets assigned a value. If it is unclear, there is an explanation of assignment following.

BIG = 50	Gives BIG a value of 10.
BIG	Shows the value of BIG .
LITL = BIG - 45	Gives LITL the value of BIG - 45.
LITL	Shows the value of LITL .
LITL = LITL + 1	Gives LITL the old value of LITL plus 1.
LITL	

Notice that the same variable can be used on both sides of the = sign. To think how the computer works with this, imagine

NEW VALUE = OLD VALUE + CHANGES

Remember that the computer looks first at the right side of the equal sign. It uses all the current values of the variables to evaluate the expression to the right of the equal sign. A legitimate expression is a sequence of variables, constants, and functions which can be evaluated to a value (single value [scalar] or row of value [array]). Once the entire expression to the right of the equal sign has been evaluated, it takes the result and puts it into the variable named on the left of the equal sign.

To avoid confusion, remember that **BARHeight** and **BARColor** are *not* variables and should not be followed with an =. **BARH** and **BARC** are functions that translate the values that follow them into heights and colors on the graph.

-10 Using Variables

There are a few clues to using variables that should become a bit more clear as you work through this manual. Just to make you aware of these clues, we'll state them here, and give a few samples.

The first is that you should always be able to tell what value each variable will have. This process is called tracing — and it can be done by hand or by the computer. Here is a set of statements which trace the value of a variable as it changes.

COWS = 24	Assigns value 24 to COWS .
COWS	Shows current value of COWS .
COWS = 22	Assigns new value to COWS .
COWS	Shows current value of COWS .
LION = 1	Creates a variable called LION , with value 1.
COWS = COWS - LION	Subtracts value LION from COWS , and assigns result to cows.
LION	Note LION did not change.
COWS	Value of COWS did change.

The tricky line above is the one that reads, **COWS** = **COWS** - **LION**.

Chapter 1: Getting Started

What it means is that the computer must take the values it has for COWS and LION, use them to compute a value, then give that value to COWS. Notice that = does not mean the same thing that it does in math, that the two things are equal. The = sign tells the computer to find the value of all the variables and numbers on the right of it — then give that value to the variable on the left of the = sign.

It's a good idea to use names which remind you of the thing you want to represent. It's easy to forget what X, Y, Z, and W3 stand for in a program. And if you forget what they mean, they you can become confused quickly. Use variables named TIME, DOGFoOd, CALOries, or WAGEs to remind you what they stand for. Also, if you use a variable named X (although legal), it is easily confused with the x sign.

The computer can only use the first four letters of any keyword or variable name. That's why we print the first four letters in bold letters in the manual. The following letters are only a reminder of what the first four letters are short for. You need only type in the first four letters of any words. But it is important that the first four letters are different from any other variable name. For example, SAIL and SAILor would be the same to the computer, since the first four letters of both words are the same.

Finally, the first letter of a variable name has to be an alphabetic character, that is, a letter from A to Z. The names in the example below show you both the right and wrong way to call variables.

WRONG	RIGHT	REASON
4WAY	WAY4	A variable name can't start with a number.
#TWO	NTWO	A variable can't begin with #.
.TOP	TOP	A period is illegal in a variable name.
BARH	BARI	A keyword cannot be used as a variable name.

-11 Precedence

When more than one function is used in an expression, different orders of evaluation could cause the generation of different results. For example, the expression "3x2*2" could result in a value of 12 or 36.

Precedence rules define a consistent order of evaluation and resolve ambiguity. The result of the previous expression in APL/S is 36.

The three APL/S precedence rules are:

Rule 1: Functions are evaluated left to right, except for the two-argument functions, +, and -, which are evaluated last in the same left to right order.

Input	Analysis	Result
3 x 5 MAX 10	15 MAX 10	15
4 - 2 x 3	4 - 6	-2
3 - 4 + 3	-1 + 3	2

Rule 2: Computations within parentheses are performed first.

Input	Analysis	Result
$3 \times (5 \text{ MAX } 10)$	3×10	30
$(4 - 2) \times 3$	2×3	6
$2n\text{LOG } (4 \times (2 \times 3))$	$2 \text{ LOG } (4 \times 8)$	
	$2 \text{ LOG } 32$	5

Rule 3: The entire righthand side of an assignment statement is evaluated before the assignment is made.

Input	Analysis	Result
$2 \times A = 1 + 2$	$2 \times A = 3$	
	2×3	6

-12 Row Variables

Variables can take the value of more than one number. There are things we'll call row variables which can take a whole row of values. Key in the following lines to get an idea of how they work.

EVEN = 20 40 60 80	Gives EVEN 4 values.
ODDS = 11 31 51 71	Gives ODDS 4 values.
BARH EVEN	Shows EVEN .
BARH ODDS	Shows ODDS .
EVEN + ODDS	Shows sum of EVEN and ODDS .
EVEN + EVEN	Shows sum of EVEN and EVEN .

The order the values of row variables are typed in is very important. Just to see how important, key in the following statements.

BARColor 2 4 6 8 10	Sets colors for 5 bars.
UP =10 20 30 40 50	Gives row variable UP 5 values.
DOWN =50 40 30 20 10	Gives DOWN 5 values.
BARHeight UP	Shows UP .
BARHeight DOWN	Shows DOWN .

Can you make a row variable called **FLAT**, with all 5's? Show **FLAT** as a **BARHeight**. What happens if you use **UP** as **BARColor**? What happens when you use **DOWN** as **BARColor**? What happens if **FLAT** is the **BARColor**? Show the values of each of your three row variables.

-13 Computation with Row Variables

You can do computation with row variables, much as you can with regular variables. Just remember that when you operate with a row variable, you work on every value in the row.

Chapter 1: Getting Started

Here are a few simple examples.

STOK = 23 25 11 18	Stocks' current value.
BARC 8 2 4 7	Sets the bar colors.
BARHeight STOK	
BARHeight (STOK + 20)	
BARHeight (STOK + 40)	Stocks appear to be going up!
BARHeight (STOK + 30)	
BARHeight (STOK - 10)	Market crash.

Notice that addition and subtraction affect all the stocks the same way. Can you multiply the price of the stocks by 2, and show it on the bars? Divide the price of the stocks by 10 and see if they show on the bar graph.

You can think of the operation this way:

YOUR STATEMENT	VIDEOBRAIN DOES					RESULT
STOK + 20						
	STOK	23	25	11	18	
	+20	+20	+20	+20	+20	
	<u>STOK+20</u>	<u>43</u>	<u>45</u>	<u>31</u>	<u>38</u>	43 45 31 38

But we all know that stocks don't gain and lose value at exactly the same rate. Some go up, others go down. In order to show different gains, we can add a row variable with the different gains to the row variable **STOK**.

In the example below, be careful of the - sign. Notice that the cents sign (¢) makes a negative sign that is above the center of the line (¯). The minus sign (upper case G) makes a minus sign that is in the center of the line. You need to key the negative sign (¯) to indicate a negative number. This is very different from the minus sign (-), which tells VideoBrain to subtract one number from another.

STOK = 23 25 11 18	Beginning stock prices.
BARHeight STOK	
GAIN = 5 ¯ 3 10 6	How much each gained or lost.
PRICe = STOK + GAIN	New PRICe beginning price + gain.
BARHeight PRICe	Shows new PRICe .
PRICe	Shows each value in PRICe .

When you add a row variable to another row variable, think of adding the first value of one to the first value of the other, and so on. If you were doing it on paper, it would look like this:

YOUR STATEMENT	VIDEOBRAIN DOES					RESULT
PRICE = STOK + GAIN	STOK	23	25	11	28	
	+GAIN	+ 5	+ ¯3	+10	+ 6	
	PRIC	28	22	21	34	PRICE = 28 22 21 34

Using this same idea, can you use a row variable called `INFLation` to increase the `PRICe` of the stocks by 5%, 8%, 2% and 14% each? Give the results to a variable called `GROWth`. Subtract a broker's fee of 10% from each of the stocks, and give the results to a variable called `NETWorth`.

-14 Input/Output

Input and output methods on a computer are called I/O for short. They refer to ways you get values into (input) and out of (output) the computer.

There are two ways to input to your VideoBrain. One way is with the keyboard. The other way is with tape, using the `LOAD` command explained in Chapter 5. The *key-board can be used in immediate mode, as we have been using it. It can also be used to input data into a program, as described in Chapter 2.*

There is only one way of outputting your program results, and that is through the television screen. There are two ways of doing this, however. The easiest way — to display the value of a variable by entering its name as a line all to itself. The other way is by showing the value as a bar height using the `BARHeight` command. You have been using both modes of output in a number of examples throughout this chapter.

`BARHeight` only shows numbers as large as +64, and as small as -64. If there is a bigger value than 64, it is shown only as tall as 64. You can rescale values with the utility program `SCALE`, in Chapter 6.

Chapter 2: Beginning Programming

Until now, you have been only using the computer in immediate mode. VideoBrain immediately does whatever the line you typed in tells it to do. For example, when you type 2+2 and then press <NEXT>, the computer immediately executes your command and the number 4 appears on the screen.

There is another mode you will now learn about called edit mode which is used to create and modify programs. Edit mode allows you to type a set of commands and then save them for execution at some future time. An organized group of these commands is called a program.

-1 Your First Computer Program

A computer program is an organized list of instructions that the computer does one at a time. In many ways it is like a recipe for cooking. Directions must be clear and concise, and they must follow a specific order. Most importantly, a program must have a precise goal: Exactly what do I want my program to do?

Let's start off with a simple program. This program will display the answers to the following series of multiplications, 2x1, 2x2, 2x3, 2x4, 2x5, and 2x6. We now know our goal. If we do things properly we will get the numbers 2, 4, 6, 8, 10 and 12 displayed on the screen.

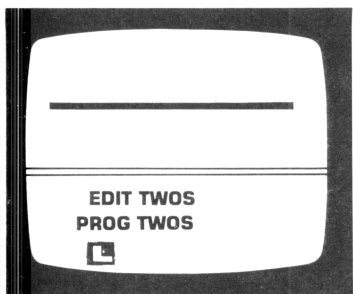
Every program must have a program name! Let's call this program TWOS, since it multiplies by 2. Now type:

EDIT TWOS

and press <NEXT>. Congratulations! You have just left immediate mode and entered edit mode. Edit mode is exclusively used for creating new programs and making changes in old programs.

Notice that VideoBrain took your line, EDIT TWOS, and added a line after it, PROG TWOS.

Your screen now looks like this:



Notice that the cursor is black with a white “L” in the middle of it. This is just the reverse of what it looks like in immediate mode — white with a black “L” in the middle of it. Looking at the cursor is the easiest way to tell whether you are in immediate mode or edit mode.



The next thing to notice on your screen is that the line “PROG TWOS” appears directly under the command “EDIT TWOS” which you just typed. PROG TWOS is short for PROGram TWOS. The computer typed this for you. It is the title of your program and it means that everything you type on the keyboard will become part of PROGram TWOS, until you type ENDEdit and get out of edit mode.

You must now type in the commands you wish the computer to execute. Remember, our goal is to display the answers to the following multiplications: 2x1, 2x2, and so on until 2x6.

Press the <SHIFT> key, and notice that the cursor now says “U” for upper case (the numbers are upper case). Type

2x1	Press <NEXT> after each line.
2x2	
2x3	
2x4	
2x5	
2x6	
ENDProgram	Ends program.
ENDEdit	Puts you back in immediate mode.

The word ENDProgram tells VideoBrain that only the material after PROGram TWOS and before ENDProgram is in program TWOS. Since you have finished editing, there is no longer any reason to remain in edit mode. So you typed ENDEdit, and got out of edit mode and back into immediate mode. Notice that the cursor is now white with a black letter in it, rather than the other way around. This confirms that you are in immediate mode.

Also, in edit mode, VideoBrain did not execute 2x1 immediately. Instead it stored this command in its memory for execution later.

You may wonder at this point what happened to PROGRAM TWOS. It is now stored in VideoBrain's memory, waiting to be called and used as you wish. To execute PROGRAM TWOS, type

TWOS

and watch the screen. If you did everything correctly, you should see the answers to the multiplication problems appear on your screen. If your program did not work push (MASTER CONTROL) and start over again. The most common error people make is to type the letter "X" instead of "x", the symbol for multiplication, which is above the letter "T" on the keyboard.

Would you like to see it again? If so, type

TWOS

Each time you type the name of a program, VideoBrain looks into its memory, finds the program, and then executes it. The computer will remember your program as long as you don't push (MASTER CONTROL) or don't turn the machine off. In Chapter 5 you will learn how to save your programs on tape.

-2 Making Changes in Your Program

You may wish to make changes in a program that is stored in memory. This is called editing a program, and is done in edit mode. As an example, let's try to make the following changes in PROGRAM TWOS:

Erase the line 2x2

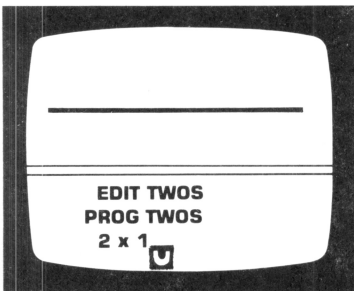
Replace the 2x5 with 2x22

Insert the line 2x2 before the end of the program

In order to make these changes, you must leave immediate mode and enter edit mode. Type

EDIT TWOS

You are now in edit mode. Your screen should show the lines PROG TWOS and 2x1, below your edit command.



Chapter 2: Beginning Programming

Notice that the cursor is on the line which says 2x1, and is directly underneath the blank space ending that line.

To see the next line, press <NEXT>. Keep pressing <NEXT> until you come to the last line of the program, ENDProgram.

How to erase 2x2. In order to modify a line, you must return the cursor to that line. Press the blue <PREVIOUS> key and VideoBrain lists all lines of the program back to the one just before the line you were on. Press <PREVIOUS> again, and you will end up one line earlier in the program. Keep pressing <PREVIOUS> until the cursor is on the line you wish to delete: 2x2. Now press the blue <ERASE> key. Notice that a white box appears at the end of the line. This means that the line is erased from memory, even though it is still visible on your screen.

How to replace 2x5 with 2x22. Press the <NEXT> key until the cursor is on the line 2x5. Press the <BACK> key until it erases the number 5. Now type the new number, 22, in its place. Press <NEXT> and the change is complete.

How to insert 2x7 before the end of the program. The cursor is now on 2x6. Press the blue <SPECIAL> key. This opens up a blank line immediately beneath the one you are on. Now type 2x7, remembering to press <NEXT>. You have now inserted 2x7 after 2x6.

You have made all the changes you want, and would like to get out of edit mode and back into immediate mode. To do this, space down until your cursor is after the ENDProgram line. Then type ENDEdit. Press <NEXT>, and you are back in immediate mode. All of the changes you have made to TWOS are now in memory.

To run your new version of TWOS, type
TWOS

You should now get a new set of new numbers on your screen: 2, 6, 8, 44, 12, and 14.

A note about names. When you named your program TWOS, you did it because that was the name in the book. You can make up names for your other programs, but you may not use as a program name any of the keywords (for example, EDIT or BARColor). They are reserved for VideoBrain, and can not be used to name either programs or variables (see Reserved Words in Appendix H). Also, if you already have a variable in memory with the same name as the one you want for your program, VideoBrain will not allow you to name your program with the variable name.

-3 Edit Command Summary

Here is a short summary of commands and keys which you will find useful in editing your program.

EDIT (program name)	Creates a program with your choice of name, and puts the VideoBrain in edit mode. If a program already exists with this name, this command will allow you to make changes in that program.
----------------------------	--

ENDProgram	This must always be the last line of your program.
ENDEdit	This is usually typed after the ENDProgram line, although it may be inserted anywhere to get you out of edit mode. It returns you to immediate mode. If you type ENDEdit on an empty line in the middle of a program, the ENDEdit line will not become part of the program.
<NEXT>	Displays the next line of the program. It can be typed after each line of a program to have it entered into the memory.
<PREVIOUS>	Displays the line immediately before the line your cursor was on. It can be typed after each line of a program to have it entered into the memory.
<SPECIAL>	Opens up an empty line below the line the cursor is on. It can be typed after each line of a program to have it entered into the memory.
<BACK>	Moves the cursor one space back on a line. It erases any character it backs over.
<ERASE>	Erases the line the cursor is on from computer memory. A white box will appear after the deleted line, even though the line still shows on the screen.

-4 Counter and Assignment

The PROGRAM TWOS which you just finished did not use any variable names. The next program you will now do uses a single variable name to count from one to four. The program may look funny to you, because the = connects two things that are not equal. But don't forget what = means in APL/S: = means assign the value of the expression on the right to the variable to the left of =.

EDIT ADD

NUM = 1	Gives NUM the value of 1
NUM	Shows current value of 1.
NUM = NUM + 1	Gives NUM the old value of ONE , plus 1.
NUM	
NUM = NUM + 1	Gives NUM the old value of ONE , plus 1.
NUM	
NUM = NUM + 1	Gives NUM the old value of ONE , plus 1.
NUM	
ENDEdit	

Chapter 2: Beginning Programming

Run this program. It shows you how the value of the variable NUM changes through assignment. Notice that NUM could be used to count from 1 to 100 – without using another variable name. This concept is essential to programming, and most programs that work with numbers are dependent on it. It is very important to keep in mind what happens with the assignment operator.

YOUR COMMAND	WHAT VIDEOBRAIN DOES	RESULT
NUM = NUM + 1	1. Looks up the current value of NUM. 2. Adds 1 to NUM. 3. Gives NUM the value of the expression.	NUM is 1. 1 + 1 NUM is now 2.

-5 The KEYBoard and “ ” Commands

In order to write more practical programs than TWOS and ADD, you must be able to enter data into the program while it is running. The KEYBoard and “ ” commands make this easy.

In immediate mode, type
“HOW ARE YOU” Make sure you type the quote marks.
“2x2”

The computer mimics exactly what you type between the two quotation marks. Notice that VideoBrain does not compute “2x2” when it is in quotation marks – it just types the characters. It will print anything you type between two quotation marks. The only character that can not be placed between quotes is the quotation mark itself.

The KEYBoard command is VideoBrain’s way of asking you for information. In immediate mode, type

AGE=KEYBoard

A question mark will appear on your screen. The computer is asking you to give it the value of the variable AGE. Now type in your own age (you must tell the truth!) and press <NEXT>. The variable AGE now has the value you gave it from the keyboard. To see the value of age, type

AGE

and the value of your AGE is displayed. It is no longer a secret.

Any time you wish to have the computer ask you for a value to be assigned to a variable, use the KEYBoard command. A question mark will be displayed on the screen, and the user must input a number. Upon pressing <NEXT>, the number is entered into the variable assigned KEYBoard. It is usually best to use a short statement in quotes before a KEYBoard command, so that the user knows what he or she is supposed to input. The short quote, with the question mark, is called a “prompt” in computer lingo, because the computer is prompting you for a response.

-6 Using Prompts in Programs

We will now write a program that is more practical than your first two programs. It will compute the weekly salary of a business firm's employees. More specifically, it will take the number of hours worked by an employee and multiply it times his hourly wage. The result is his weekly pay (except for deductions).

We know the program's goals. Next, we must develop a set of clear directions which will accomplish our goal. We write them out in plain English, then translate them into APL/S. These are the steps we might use in the salary program:

GOAL:	Compute an employee's weekly pay.
FIRST:	Ask how many hours the employee worked during the week.
SECOND:	Ask what the employee's hourly wage is.
THIRD:	Multiply the hours and wage rate together and display the answer. This is his weekly gross pay.

Now, let's translate these steps into APL/S. In the program, the comments on the right will show how the lines of the program implement the steps above.

EDIT PAY	The name of our program is PAY.
PROGRAM PAY	
"HOW MANY HOURS"	Labels the coming question.
HRS=KEYBoard	Takes input from the keyboard and assigns it to variable HRS.
"HOURLY WAGE"	Labels the coming question.
WAGE=KEYBoard	Takes keyboard input and gives it to the variable WAGE.
"WEEKLY PAY"	Labels the data to come.
HRS x WAGE	Display the value of HRS x WAGE.
ENDProgram	End of program PAY.
ENDEdit	Gets out of edit mode.

From now on, you will have to remember to type ENDEdit after the program to get out of edit mode. We will not be putting that in the directions, even though you will need to do it each time.

Notice how the program reflects the three stages of our thinking: input the hours, input the wage rate, then display the answer.

The program is now in memory and VideoBrain is in immediate mode. To run the program, type

PAY

and pay close attention to your screen. Answer the questions, and see if it computes your pay correctly.

NOTE: the dollar sign (\$) is not legal as part of a number, either as input, or used in a program as part of a number. When the program asks you how much the weekly wage is, do NOT type \$11.50 — instead type 11.50. If you use the dollar sign, you will get a VALUE error.

To figure the pay for another employee, type PAY again. This calls the program from memory again. How much is someone's salary who works 37.25 hours and earns 8.35 dollars per hour?

-7 A Program of Your Very Own

You now know enough about VideoBrain to write your own short program. Give this problem a try:

GOAL: A program that will compute your major monthly expenses. The program will prompt you for the cost of food, housing, car, etc. It will add all these together and display the final total.

You now have a clear goal. The next step is to think through the problem in a clear series of steps. What would you do first, second, third, and so on, to solve the problem? Write these steps down in plain English. Then translate each step into APL/S.

Here are some hints:

HINT #1: Your first program step might be something like this: Ask how much is spent on food, and assign it to a variable called FOOD.

HINT #2: At the end of your program you will want to compute and display your total expenses. Make sure that all the values you input are totalled in the final amount.

HINT #3: In order to type this problem, or any program into VideoBrain, you must be in edit mode. This can be done by typing EDIT, followed by the imaginative name of your program. The last line of the program must be ENDProgram, followed by an ENDEdit, to return you to immediate mode.

Have fun writing the program. If you make mistakes and it doesn't work don't worry. The remainder of this book is filled with dozens of examples of how to write programs, how to correct errors, and how to think through programs. You will have many more opportunities to develop your programming skills. Almost no one's first computer program works correctly the first time. If yours succeeds, pat yourself on the back for being one of the select few. If yours failed, relax and consider yourself in good company, many of whom are now professional programmers.

-8 Three Kinds of Words

Although you may not have been conscious of it, you have been using three different types of words with your VideoBrain. One type is variables. A second type is keywords. And the third type is system commands. They are summarized in this table:

VARIABLES	KEYWORDS	SYSTEM COMMANDS
EXAMPLES: TOTL, COWS, MOM, FMLY	BARHeight PROGRAM BARColor, +, =	LIST EDIT ENEdit
FUNCTION: Name a place to store a value, or several values.	Special words or symbols that can be used in programs or immediate mode to relate values.	Tell the system what to do. They change the state of the system.
SPELLING: You can make up your own 4-letter combinations.	Must be spelled exactly right — or you will get an error message.	Must be spelled right or nothing happens.

Keywords occur all the time in programs and as you use the computer in immediate mode. These are the building blocks of programs and computational routines. When they are used, keywords always start a new logical line. The variables change from program to program — they should be named so you can remember what the stored values mean. Variables can be spelled in almost any way — within the limits stated in Chapter 1. And the system commands let you work with programs and manipulate the variables in memory. Some system commands allow you to LOAD your programs and variables on tape or ERASE them entirely. Others tell the system to start saving a program in edit mode, or that edit mode is ended. <MASTER CONTROL> is actually a system command, as are all the keys used for making corrections. The distinctions between these three kinds of words may not be terribly clear right now, but if you try to keep them in mind, it will be much easier to understand what is happening with your VideoBrain.

Chapter 3: Flow of Control

So far, the programs and statements you have been using have been executed in a series, from first to last. The small programs you have seen so far all work in a similar way. This chapter will explain how to control the flow of a program so that some commands are executed while others are not.

-1 Blocks

Any series of statements that is executed in order is called a block. The smallest block is a single statement — it is one statement executed in order from first to last. When you type a program, there are markers for the beginning and end of a block. The entire program is a block, and the beginning marker is the keyword **PROG**ram, with the ending marker **END**Program.

Here is a simple program. We can draw a line from the beginning marker to the ending marker, to show the group of lines that is executed in order.

```
PROGram RISE  
  "THIS PROGRAM GOES"  
  " IN ORDER "  
  BARHeight 10  
  BARHeight 20  
  BARHeight 30  
  BARHeight 40  
  BARHeight 50  
  BARHeight 60  
  "END OF BLOCK"  
ENDProgram
```

Run program **RISE** a time or two. It is a single block of statements — a series of commands which are executed in order.

While it doesn't make a lot of sense to mark a block right now, by the end of the chapter it will become very important. In the next section you get into a situation where you don't want all the steps executed in order, and where understanding a block is very significant.

-2 IF .. THEN .. ELSE .. ENDIf

Suppose you are coming up to a traffic light. You look at the light, and without being aware of it you make a decision:

```
IF the light is green  
THEN go ahead  
ELSE get ready to stop.
```

The color of the light will control your actions, unless you want to get a ticket.

Chapter 3: Flow of Control

Notice that if this were a program, you would not want to execute both commands — one to stop and the other to go ahead. You aren't going to do both, just one OR the other.

The same example, as a computer program, would be like this.

PROGRAM CONDitional	
LITE = KEYB	Lets you put in a number
IF LITE GT 0	If the number is Greater Than 0
THEN "LITE IS POSITIVE"	then say "Positive"
ELSE "LITE IS NEGATIVE"	Otherwise, say "Negative".
ENDIF	If statement is done.
ENDP rogram	

Run the program several times. Notice that whenever you put in a negative number (like -4), the computer skips the first statement, and only executes the statement which says "LITE IS NEGATIVE."

Control statements such as IF . . THEN . . ELSE (and the WHILE statement covered later in this chapter) are only able to be used as part of a program. They cannot be used in immediate mode, since they only make sense when used together with other statements.

-3 Conditionals

The controlling part of the traffic light example is the color of the light. This is called the condition. If the condition is true, then go ahead, otherwise stop. Of course, VideoBrain can't look at traffic lights, but it can look at other conditions. A condition for VideoBrain might be "LITE GT 0" (the variable LITE is Greater Than 0). All conditionals are either true or false. If the condition is true, then one block of statements is executed. If the condition is false, then another block of statements is executed.

There are six conditionals. Each of them is really an abbreviation for a longer phrase.

EQ (EQual to)	NE (Not Equal to)
GT (Greater Than)	LT (Less Than)
GE (Greater than or Equal to)	LE (Less than or Equal to)

Notice that the equals sign (=) is not used to mean EQual in a conditional. The equals sign tells VideoBrain to assign a value to the variable to the left of the sign. When you want to see whether two values are equal, you must use the EQual keyword, and not the = sign.

-4 Example

Let's see how we might use one of the conditionals above to make up a simple program.

PROBLEM: Make up a program which will turn a single bar red if the number input is below zero, and turn that bar green if the number is above zero.

THINKING IT THROUGH: Here are the steps you might use if you were doing it by hand.

- 1) Start out with a yellow bar.
- 2) Ask for the number.
- 3) If the number is below zero, then color the bar red, otherwise, color the bar green.

Here is a sample program that does this:

PROGRAM BARS	
BARHeight 50	Puts up single bar
BARColor 14	Colors bar yellow
NUMBER = KEYB	Input any number
IF NUMBER GE 0	If number is greater than 0
THEN BARColor = 2	then turn bar green (2)
ELSE BARColor = 4	Otherwise, turn bar red (1)
ENDIF	End IF statement
ENDProgram	

VARIATIONS: Run the program several times. See if it works. Can you change the condition so that the light turns green only if the number is greater than 10? Change the condition so that the light turns green for numbers smaller than zero, and turns red for numbers 0 or larger.

-5 Bigger Blocks

When you want to do two or more things on each condition, you can have as many statements as you want in each block. Here is an example program:

```

EDIT LET2
NUMBER = KEYB
IF NUMBER GT 0
THEN "FIRST BLOCK"
  "N BIGGER THAN 0"
  BARColor 2
  "END FIRST BLOCK"
ELSE "SECOND BLOCK"
  "N LESS THAN 0"
  BARColor 4
  "END SECOND BLOCK"
ENDIF
ENDProgram

```

Chapter 3: Flow of Control

You can do as many things as you want to do between THEN and ELSE. You can also have as many statements as you want between ELSE and ENDIf. Each group of statements is a block, because it will execute in order, from first to last.

In the LET2 program above, notice that all the statements in a block are indented the same amount. This shows you that they are executed in a series together. The indentation is used to show which statements are in a group. It is strongly recommended that you use indentation when writing your programs. You may type in the indentations, but when the program is shown in edit mode, the indentations disappear.

-6 New Lines for THEN and ELSE

Following an IF . . (condition) . . line, the keyword THEN must start a new line. If it doesn't start the next line, you will get an error statement, SYNTAX 93.

The keywords IF, ELSE, and ENDIf must also begin on a new line.

-7 Omitting ELSE

The keyword ELSE is optional in an IF . . THEN . . ELSE . . ENDIf statement. If you don't have anything ELSE to do, then just leave out the ELSE. You can not leave out the ENDIf, however. The following program gives a good example of an IF statement without an else.

```
EDIT ALRM
"KEY IN A #"
NUMBER = KEYB
IF NUMBER EQ 10
THEN "ALARM"
  BARHeight 64 64 64 64 64 64
ENDIf
ENDProgram
```

There is no ELSE in the above program, and one is not needed. Only if the number is 10 is there an alarm. Otherwise, the program can just continue. Notice that there is an ENDIf, to let VideoBrain know that the IF statement is over.

-8 Multiple Conditions and NOT

Sometimes more than one condition has to be checked in an IF statement. In such cases, the keywords AND and OR can be used to join conditions.

Here are some examples:

```
IF (COST LT PURS) AND (COST LT 100)
IF NOT ( TIME EQ 0) OR (HORSe GT 64)
```

When more than one condition is being used, each of the conditions may be put inside parentheses (). As you can see, NOT is put outside the parentheses to negate the condition in the parentheses.

PROBLEM: Write a simple budget program. Put in the amount of money you have, and the cost of three things. If your things cost too much, borrow the amount you need plus \$10. If your money will cover them, pay the bills and compute the change.

THINKING IT THROUGH: Here are the steps that you would do if you were going to do the job by hand:

- 1) Find the amount of cash on hand.
- 2) Ask the amount of the expenses.
- 3) Find the total of expenses.
- 4) If your cash is not enough, then borrow.
- 5) Otherwise, pay the bills and see what's left.

```

PROGRAM PAYBills
"CASH ON HAND?"
CASH = KEYB
"FOOD, RENT, CAR?"
FOOD = KEYB
RENT = KEYB
CAR = KEYB
TOT = FOOD + RENT + CAR
IF CASH LT TOT
THEN "BORROW "
  BORROW = TOT - CASH + 10
  BORROW
ELSE "CASH LEFT "
  CASH - TOT
ENDIF
ENDProgram

```

VARIATIONS: Can you graph the costs of each item, along with the cash? Graph the total red if you have to borrow, and graph it green if you have enough money.

-9 WHILE . . DO . . ENDWhile

There are many times that you want to repeat a line over and over again. In APL/S, this is done with a WHILE loop. The computer keeps repeating the lines (the block of statements) between DO and ENDWhile until the condition on the WHILE line is false. Then it stops executing the WHILE loop and goes on.

Chapter 3: Flow of Control

A simple program shows you how a counting loop goes on.

```
PROGam GOOFy
COUNT = 1
WHILE COUNT LT 100
DO COUNT = COUNT + 1
  COUNT
ENDWhile
"DONE WITH COUNT"
ENDProgram
```

Run the program and see how long it takes VideoBrain to count to 100. When COUNT is not less than 100, the WHILE loop stops and you get the message "DONE WITH COUNT".

Insert a line in the program which says "STILL COUNTING" every time the counter goes through another loop. Can you modify the loop with an IF . . THEN . . ENDIf statement so that it only shows its value every time it gets to a multiple of 10? A word of caution: IF cannot follow DO on the same line. IF must be on the next line.

The variable which is in the WHILE condition is called the control variable. The control variable has to be a value that makes the condition true at first for the computer to get into the WHILE loop. And the control variable has the change inside the WHILE loop, or the condition will always be true and the WHILE loop will never exit. The following program has one of these conceptual errors. Tell which one it is, and see if you can fix it.

```
EDIT MORE
ONE = 1
MORE = 1
"BEGIN ONE MORE"
WHILE MORE LT 50
DO MORE = 1 + 1
  MORE
  ONE = ONE + 1
  ONE
ENDWhile
"ONE MORE END"
ENDProgram
```

If you have trouble following a program while it is running, use the <RUN/STOP> key. If you hit the key one time, then the program will halt. When you hit the <NEXT> key after the <RUN/STOP> key, the program will begin running again. To stop the program for good, hold the <RUN/STOP> key down and you will be in immediate mode again.

A few things are important to notice in the WHILE . . DO . . ENDWhile statement. The DO has to be on the next line after the WHILE and its conditions. The conditions are the same as for IF . . THEN . . ELSE . . ENDIf covered in this chapter. Also, the loop has to end with ENDWhile, or else VideoBrain will go to the end of the program and wonder how many statements are inside the loop — and where it ends so you get an error statement.

-10 EXIT command

Often, during a WHILE loop, you would like to have a way to get out of the loop if something unusual happens. Suppose you wanted to make up a program to keep score at a baseball game. You certainly couldn't anticipate whether or not it was going to rain. In the program below is a statement that checks to see if the number of errors input during an inning is 100. The value 100 would be a flag to show that it was raining, and would cause an early EXIT from the loop. Look at the following program to see how EXIT is used to stop the scoring early if necessary.

```

PROGm BALLgame
INNIg = 1
TRUN = 0
THIT = 0
TERR = 0
WHILe INNIg LE 9
DO "INNING "
  INNING
  "KEY RUNS HITS ERRS"
  RUNS = KEYB
  HITS = KEYB
  ERRS = KEYB
  IF ERRS EQ 100
  THEN "IT IS RAINING"
    EXIT
  ENDIf
  TRUN = TRUN + RUNS
  THIT = THIT + HITS
  TERR = TERR + ERRS
  "TOTAL R, H, E ARE"
  TRUN
  THIT
  TERR
ENDWhile
"GAME IS OVER"
ENDProgam

```


Chapter 3: Flow of Control

The EXIT statement does not get you out of the whole program — just out of the particular WHILE loop that you are in when the EXIT statement is executed. When the EXIT statement is executed, it puts VideoBrain right after the next ENDWhile statement.

There are some problems with the ballgame program. It only keeps score for one team. Can you make it keep score for two teams? Also, the program won't help us much if the score is tied at the end of the ninth inning. How can you change the program so that it keeps going until somebody wins?

-11 Nested Structures

WHILE, THEN, and ELSE *blocks* may contain other WHILE and IF statements. The last "root of a number" example below uses nested statements to deal with negative values.

```
PROG LAST
"ENTER NUMBER"
IN=KEYB
N=5
I=2
WHILE I LE N
DO S=1
  OUT=IN
  IF OUT LT 0
  THEN OUT=-1N-1N
    IF I MOD 2
    THEN S=-1
    ELSE "IMAGINARY"
    ENDI
  ENDI
  OUT*=IxS
  I=I+1
ENDW
"END OF ROOT EXAMPLE"
ENDP
```

WHILE condition is true
Execute this block

IF condition is true
THEN use this block
IF second condition is true
THEN use this statement
ELSE use this statement
End of inner IF
End of outer IF

End of WHILE; go back to WHILE and repeat

Chapter 4: Row Variables

If you have worked through the previous chapters of this book, you have already been introduced to row variables. This chapter will provide you with a quick review of what you already know about row variables, then it will show you new and powerful ways to use row variables in your programming.

Row variables are variables that contain a row of values. Key in:

```
COST = 8 5 6
AGE = 20 30 45
```

CAUTION: Don't use <MASTER CONTROL> throughout this chapter or it will erase the values of COST and AGE which are used in many examples on the following pages.

Both COST and AGE are row variables because they each contain a row of values. Each value in COST might refer to one item in a store. Each value of AGE might refer to the age of a certain person.

You can add, subtract, multiply, and divide row variables in the same way as single valued variables. Key in:

```
COST x 2      Multiplies each value in COST by 2.
2 x COST      Does the same.
COST - 1      Subtracts 1 from each value of COST.
5 + AGE       Adds 5 to each value in AGE.
AGE ÷ 3       Divides each value in AGE by 3.
AGE
```

When you command COST x 2 the computer does the following:

YOUR COMMAND	THE COMPUTER DOES	RESULT
COST x 2	COST = 8 5 6 x2 x2 x2 16 10 12	16 10 12

From the last command above, notice that adding 5 or dividing by 3 did not change the value of AGE itself. They did not assign new values to AGE — they merely used AGE to compute results. In order to change the value of the row variable, you have to assign it new values. While it may sound confusing, key in the example below, and you will understand.

```
AGE           Shows current value of AGE
AGE + 5       Shows AGE + 5
AGE           Notice AGE didn't change
AGE = AGE + 5 Gives AGE value of old AGE + 5
AGE           Shows new values of AGE.
COST
COST x 2
COST
COST = COST x 2
COST
```

Chapter 4: Row Variables

Using the assignment symbol =, try to take a dollar off the cost of each item in COST. Add ten years to everyone's age in AGE.

-1 Operations with Two Row Variables

AGE - 4 is an example of doing arithmetic with a row variable and a single number. You can also do arithmetic with two row variables at the same time. Key in:

```
CATS = 3 5 7
DOGS = 1 10 100
CATS + DOGS
```

The command CATS + DOGS is an example of arithmetic using two row variables. The diagram below shows you how the computer pairs the values in the two row variables while doing the operation.

YOUR COMMAND	WHAT THE COMPUTER DOES	RESULT
CATS + DOGS	$\begin{array}{r} \text{CATS} = 3 \quad 5 \quad 7 \\ \text{DOGS} = \begin{array}{r} +1 \\ \hline 4 \end{array} \quad \begin{array}{r} +10 \\ \hline 15 \end{array} \quad \begin{array}{r} +100 \\ \hline 107 \end{array} \end{array}$	4 15 107

Now try these. Look closely at the results you get. Notice how the VideoBrain operates on each pair of values.

```
CATS
DOGS
CATS x DOGS
(CATS x DOGS) + 3
(CATS x DOGS) x DOGS
CATS
DOGS
PIGS = 1 2 3 4
PIGS + CATS
```

Did you get an answer to the last command? You shouldn't have. Instead, you received an error message. Notice that PIGS has four values, while CATS has only three. The computer will not operate on two row variables which have different numbers of values. If you wish to add, subtract, multiply, or divide two row variables, they must have the same number of values. VideoBrain allows no exceptions. Change the size of PIGS, and try again.

```
PIGS = 1 2 3
CATS
PIGS + CATS
```

Adding and subtracting CATS, DOGS and PIGS may be fun but it makes little sense. You will find row variables very useful in actual practice. Here are a few typical examples:

PRC = 2.45 5.79 3.42	Prices of 3 items in a store
QNT = 5 10 18	The quantity purchased of each item
PRC x QNT	Total spent on each item
SHRS = 15 40 25	The # of shares owned of 3 stocks
PRC = 84.25 9.5 80	The purchase price of each stock
PUR = PRC x SHRS	PUR = values of stocks when bought
PUR	Show value of each stock when bought
LIST = 85.5 7 98.125	Current list price of each stock
LIST	
GAIN = LIST - PUR	
GAIN	Shows amount of gain on each stock

Notice how easy it is to work with a group of prices, quantities, or totals when you have row variables to work with.

NOTICE: Keep these values for use in the next section. Do not use (MASTER CONTROL).

-2 Reduction Operator

The symbol “/” is called the reduction operator, and is used to combine values in a row variable. The slash tells VideoBrain to operate on all the values in a row variable. In the example above, we know how much each value gained — in the variable GAIN. But we don’t know how much our total gain or loss is. What we would like to do is add all those values together and see what we have. The second line will do it:

GAIN

+/ GAIN Adds all the values in GAIN.

The reduction operator operates on all the values in the row variable to the right of the /. Try these:

RABbits = 2 4 6 8

+/ RABbits Adds together RABbit values.

x/ RABbits Multiplies values in RABbit.

-/ RABbits Does $2 - 4 - 6 - 8$

÷/ RABbits Does $2 \div 4 \div 6 \div 8$

When the reduction operator is applied to a row variable, a chain of computations starting at the left and moving to the right is executed. The computer puts in the sign you type to the left of the reduction operator, and then executes the action. The statement **+/ RABbits** produces $2+4+6+8$. This chain operation starts at the left with $2 + 4$. The answer is 6. To this answer is added the next value, 6. $6 + 6$ produces the new answer 12. To 12 is added the next value to the right, 8. $12 + 8 = 20$. So **+/ RABbits** gives 20.

Chapter 4: Row Variables

Using a reduction operator does not change the value of the row variable. If you want to change the array, you must use the = assignment operator.

RABBI s	Shows current value of RABBI
+/ RABBI s	
RABBI s	No change in RABBI's value.
RABBI s = +/ RABBI s	Assigns RABBI total value.
RABBI s	

-3 Using Individual Values of a Row Variable

So far, you have been working with row variables as a total group. But many times you would like to be able to use or change just one value in the entire row variable. In Calaveras County, California, during the late spring, the annual frog-jumping contest occurs. Suppose your VideoBrain has been contracted to display the results of a small group of frogs. Suppose you have four frogs' current jumps. Frog 3 just made a spectacular jump. Here's how to show it:

BAR Color 13 2 15 4	Sets bar colors
JUMP = 24 34 27 22	Latest jump results
BAR Height JUMP	Shows jumps
JUMP (3) = 58	Frog 3 made a super jump
BAR Height JUMP	New results

Can you make frog 2 have a jump of 55? Show it on the bar graph. Poor frog 4 slipped, and made a jump of 14. Put that jump in the JUMP row variable, and show it on the graph. Keep the frogs around — they will be needed in the next few examples.

Real frogs have names in these contests, and it seems a shame to waste the names by having to call them by number. Here's how to call them by name.

HERS hey = 1	First frog named HERShey.
HIMM alaya = 2	Second frog HIMMalaya.
WEEH auken = 3	Third frog.
YOUTH ful = 4	Fourth.
BAR Height JUMP	
JUMP (HERS hey) = 8	HERShey jumped 38.
BAR Height JUMP	Take a look.

Give HIMMalaya a jump of 42, and show it on the graph. WEEHauken got a jump of 61 off!! Put her jump in the array, and show it on the graph.

Now, when one of the frogs jumps, we can just put the name of the frog in parentheses after the row variable. Of course, the name really refers to a number, and we could always put the number in instead. But for most purposes, the name is easier to remember.

-4 The Join Function

Suppose a new frog named PRINce joins our contest. How can we put him in the row variable? Do the following example, which adds him to the JUMPers, and puts in his first jump.

PRINce = 5

JUMP = **JUMP**,58 Add first jump of 58 to JUMP

BARHeight **JUMP**

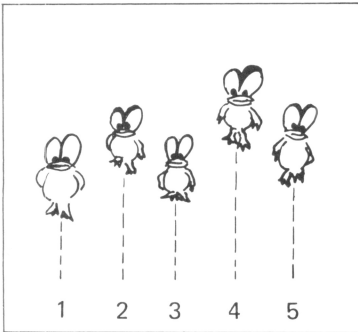
JUMP(**PRINce**) Shows PRINce's first jump

The comma (,) is the join function.

Add a sixth frog to the contest named LEGS. Enter her first jump of 27 to the group. You might want to give her a barcolor different from PRINce. (The table of bar colors is in Chapter 1.) Keep the frogs jumping until somebody has gone off the graph.

-5 The SIZE Function

JUMP = 5 16 6 33 14



Chapter 4: Row Variables

If somebody new came along and wanted to know how many frogs were jumping, they could look at the graph and see. But the graph only shows 7 values. Suppose there were many more frogs jumping.

The **SIZE** command tells the size of the row variable. It tells how many numbers there are in it. Type the following example:

```
SIZE JUMP           Tells how many jumpers there are
ONRS = 1 3 5
SIZE ONRS
CHEE = 2 4 6 8
SIZE CHEE
```

Notice that **SIZE** is always followed by a row variable. You can use **SIZE JUMP** just as any other number in the computer. In the example, **SIZE JUMP**, the variable **JUMP** is called the argument of the **SIZE** function.

-6 MINimum and MAXimum

It's nearing sundown, and the frogs are exhausted. **LEGS** makes her last jump, and you want to find the winner. By now the jumps are so long they're off the graph. What is the winning jump? Just use **MAX** to see what the longest jump is.

```
BARHeight JUMP
JUMP
MAX / JUMP
MIN / JUMP
```

When **MAX** or **MIN** are used with a single row variable, the slash "/" mark must be used after either one. The slash is called the reduction operator, since it reduces the many values in an array to one value.

MIN tells you the **MIN**imum (or smallest) jump. Notice that neither **MAX** nor **MIN** tell which frog made the longest or shortest jump. You have to check the **JUMP** array itself to see what number the frog was which made the longest jump. There is a utility program in Chapter 6 which will pick out the maximum or minimum value for you.

-7 Some Technical Terms

So far, we have been talking about row variables, and about numbers in those row variables. Computer people have their own words for these things, and sooner or later you may want to talk to one of these people. When you do, use the term “array” for a row variable. They mean the same things. All the row variables we’ve used so far can be called “arrays”. They happen to be a special kind of array, with one row only. There are arrays with two, three, or more rows, but you won’t need them in APL/S.

Also, the specific value in an array is named by a number. For example, we had JUMP(3) refer to the jump of frog 3. This 3 is called a “subscript”. The subscript tells which value in an array you are referring to. The subscript for STOC(11) is 11, and means that you want shown the 11th value in the array STOC.

-8 The ARRAY Function

If you want to start your own frog-jumping contest, you don’t have to go to Calaveras County. You can start your own contest right on your own screen, with the array command. Try it:

```
JUMP = ARRAY 4
BARHeight JUMP
JUMP
```

This is just a small, 4-frog affair, but you can put in more if you want to. Notice that the command ARRAY 4 makes up an array with 4 values, each one of them 0. If you had said, ARRAY 12, VideoBrain would have made up an array which had 12 values. Each value is always set to zero to begin with. As your frogs jump, you can change the values.

If your array should start with numbers that are not zero, you can add to the array when you create it. Do the following:

```
FLAT = ARRAY 4 + 20    Makes a 4-valued array of zeros,
                        adds 20 to each value
BARHeight FLAT
FLAT
```


-9 The INDEX Function

Suppose you wanted to start an array which contained the numbers 1 to 7. You can do it easily with the INDEX command. Like this:

```
WEEK = INDEX 7
WEEK
BARHeight WEEK x 8
```

Any time you use the INDEX function, you have to use a value after it. Then INDEX creates an array, with values from 1 to whatever number comes after INDEX. To create an array MONTH, you would use the line MONTH = INDEX 31.

-10 Reduction with Logical Expressions

The reduction operator / can be used with relational operators like EQ (Equal) or GT (Greater Than). Remember that the reduction operator reduces an array to a single value.

Suppose you have a PROFit array with the profits and losses from six stocks in it. What you want to do is find the number of stocks which made money. Here is how you would do it:

```
PROFit = 67 23 -15 -21 5
WINS = +/ (PROFit GT 0)
WINS
LOSS = +/ (PROFit LT 0)
LOSS
```

Notice that the effect of addition with the reduction operator on a logical expression is to count how many times the expression is true. For example, the statement +/ (PROFit LT 0) sees how many times it is true that a value of PROFit is less than 0. It does this for each value.

Here is a simple way to think about what the computer did:

+/ (PROFit LT 0)											
PROFit	67	23	-15	-21	5						
less than 0?	no	no	yes	yes	no						
result	0	+	0	+	1	+	1	+	0	=	2

The table above shows you how the APL/S computes the result of +/ (PROFit LT 0). Notice that it answers the logical question, "Is this value less than zero?" for each value of PROFit. If the answer is yes, then it tallies a 1. If the answer is no, then it tallies a 0. With the reduction operator, the 1's and 0's are added together.

See if you can write a statement to count the number of stocks with a profit greater than 20. Make VideoBrain count the stocks with a profit of 0.

-11 MAX and MIN with 2 Arrays

You have already worked with the MIN and MAX functions on a single array. MAXimum and MINimum can also compare two values, and return the one indicated. Try the following example in immediate mode:

```
2 MAX 38
4 MIN 21
38 MAX 4
```

Notice that when used with two values, MAX picks the bigger value and returns it. It will do this with constants, one-valued variables, and as you will soon see, with row variables.

MAX and MIN may be used to compare values in two arrays. Key in the example.

RAMS = 0 13 3 7	Scores in 1st, 2nd, 3rd, & 4th quarter
JETS = 7 0 3 17	Jets' scores in each quarter
RAMS MAX JETS	The biggest score in each quarter
RAMS MIN JETS	The lowest score in each quarter

Save your RAMS and JETS for the next couple of examples. Notice that when MAX or MIN are used with two arrays, you do not use the slash (/) mark.

To evaluate two arrays, APL/S compares each pair of values in them. Here is how APL/S works:

```
(RAMS MAX JETS)
      RAMS =    0 13 3 7
      JETS =    7  0 3 17
Pick the bigger number  7 13 3 17
```

You can imagine MAX (or MIN) looking up and down each pair of values, looking for the biggest (or smallest) one.

Needless to say, MAX and MIN will not work on arrays which don't have the same number of values. Neither MAX nor MIN could define a correct selection, since there wouldn't be two values to select from.

-12 The Comma as Ravel

Sometimes in a program, you don't know whether a variable will have only one value, or many. If it has many values, then you can use SIZE with it. If it has only one value, it is a scalar, and can't be normally used with SIZE.

To solve this problem, APL/S has a function called **ravel**, which is the comma. If the comma occurs before a scalar variable, it turns the variable into an array with one value. This might seem like a bit of technical wizardry, but it is helpful sometimes.

Chapter 4: Row Variables

Try this immediate mode example:

N=4	N is a scalar variable
SIZE N	Gives an empty array, which displays as a blank line
N=,N	Turns N into an array with one value
N	N still one value
SIZE N	But now SIZE gives the value 1

You can change the values.

In a program, the comma is helpful because sometimes you can't be certain that you'll be encountering a row variable. The comma doesn't change an array, but it does make a scalar compatible with row variable functions.

Here's a short program which shows the value of the comma.

```
PROGram AVERage
"INPUT NUMBERS "
NUMS=KEYBOard      Asks for keyboard input
NUMS=,NUMS          Turns NUMS into array
"AVERAGE IS"
+ / NUMS ÷ SIZE NUMS  Divides sum of NUMS by SIZE
ENDProgram
```

If you take out the line `NUMS=,NUMS`, the program works fine until you input a single number. Then it displays a blank line. With the comma in, the program averages even a single number correctly.

-13 Programming Practice

With all these tricks, there are lots of things which can be done to arrays. Here is a *small sample of some of them*. Do these, then see if you can answer the questions after the practice statements. Make sure you have the `RAMS` and `JETS` statements from the previous page in your workspace.

JQTR = +/(JETS GT RAMS)	
JQTR	Quarters the Jets won
RQTR = +/(RAMS GT JETS)	
RQTR	Quarters the Rams won

Make APL/S count the number of quarters in which the Jets and Rams had the same amount of points. Make VideoBrain total the number of points the Jets got. Make VideoBrain total the Rams' points. How many points were scored in the entire game? What is the point spread (the difference between the winner's score and the loser's score)? Who won the game? Of course, you can answer these questions by looking at the scores and using pencil and paper. But see if you can write APL/S statements which give you the answer for each one.

The following examples use arrays and scalars to form legal APL/S expressions. You should study them to make sure you understand how arrays are treated in APL/S.

Input	Analysis (Arrays are boxed for clarity)	Result
10+20 30	$10 + \boxed{20\ 30}$	30 40
10 20+30 40	$\boxed{10\ 20} + \boxed{30\ 40}$	40 60
10 20+30	$\boxed{10\ 20} + 30$	40 50
30÷10 5+20	$30 \div \boxed{10\ 5} + 20$	23 26
10 20+30 40 50	$\boxed{10\ 20} + \boxed{30\ 40\ 50}$	Value 3 error.

In the last example, the two arrays have different sizes. No meaningful results can be generated; therefore APL/S generates an error message.

Chapter 5: Saving and Storing Programs

-1 The Concept of Workspace

The portion of VideoBrain’s memory which is available for your use is called the workspace. The workspace has all the current values of your variables, as well as any programs you may have defined. To see what’s in VideoBrain’s entire workspace, just ask VideoBrain to LIST. (MASTER CONTROL) clears out the workspace entirely. ERASe allows you to erase just one variable or one program from the workspace, without touching the other variables or programs. In this use, ERASe is a command typed with four letters — not the (ERASE) key. There is a very big difference.

Work through the following example to understand how each of these features operates.

JUMP _s = 28 49 22 60	Sets up row variable
PULL _s = 2 5 9 6	Sets up another row variable
LIST	Shows entire workspace.
ERASe PULL _s	Just erases row variable, PULL _s
LIST	Shows what’s left
(MASTER CONTROL)	Wipes out everything
LIST	Shows what’s left

A copy of the workspace is transferred to recording tape when you use the SAVE command. The workspace is empty when you first turn on the machine, and as you use it in either immediate mode or edit mode, you change what is in the workspace. The edit keys are used to change lines and characters in the workspace. Any time you need to know exactly what is in the workspace, use the LIST command. And make sure you know what is in the workspace before wiping out everything with (MASTER CONTROL). A mistake could cost you hours of programming effort.

-2 SAVE

When you have worked hard to create a program, you naturally want to be able to store it somewhere so that you can call it to use at any time. In order to store programs on tape, you need the Expander 1 accessory. To give you the power to store and retrieve programs, VideoBrain has the commands LOAD and SAVE. The LOAD command tells VideoBrain to get a program from tape. The SAVE command tells it to save the entire contents of your memory on tape. SAVE stores all programs, variables, and values of those variables. When you want them back in your active memory, LOAD gets them off the tape and puts them back in your workspace.

It is extremely important to connect the Expander 1 to your VideoBrain **before** you begin to work at the computer. The process of connecting the Expander 1 to your computer may erase all of the data and programs which are in your workspace at the time of connection. You may lose whatever work you have done if you attempt to connect the Expander 1 in the middle of your work session with VideoBrain. (It is not necessary to connect the tape recorder before you begin work. This can safely be done anytime during your work session.)

Here's how to save a program.

- 1) Connect the round cord from Expander 1 to the back of the VideoBrain where it says "Accessory".
- 2) Plug two jacks into the record plugs of the Expander 1, as indicated in the Expander 1 manual. Plug the free end of the smaller jack into the REMOTE plug of the tape recorder. Plug the free end of the larger jack into the AUXiliary input.
- 3) Place a tape cassette in the recorder, and make certain that the recorder has power. You can usually use a very short cassette — a C-15 cassette is fine.
- 4) Make sure the tape is rewound, then turn the recorder on to RECORD.
- 5) Go to the VideoBrain keyboard and type the word SAVE. This will start the VideoBrain sending data from the workspace to the tape. The cursor will disappear during this process.

You can expect that the tape will take about a minute and thirty seconds to write. It takes this long to write the tape, no matter how many programs or variable values are in the workspace, since SAVE writes the entire contents of the workspace, blanks and all.

When the cursor shows up again, and you get a TAPE OK message, the data and programs should be all safely stored on tape. Use VERIfy (explained below) to check that the workspace contents were saved correctly. If you get a TAPE ERRor message on the screen, check your Expander 1 manual for troubleshooting instructions.

What does SAVE do? It stores every piece of information in your workspace. It stores every variable name, the current value of that variable — it even saves every program which is currently defined. If a variable is a row variable, it remembers all of the values the row variable has now. In essence, the SAVE command makes a copy of every piece of information the VideoBrain has access to.

-3 LOAD

To get a program from tape, the process used to SAVE the program is essentially reversed. Before LOADING some program and data, make sure that whatever you have in your workspace is all right to be erased. When you bring some data in from the tape by way of LOAD, everything you have in your workspace is destroyed — so if you don't want that to happen, make sure you save whatever you have first.

In order to load data from a tape into VideoBrain's memory, you must first adjust the volume and tone controls of your tape recorder. Each recorder is different, and therefore what is a proper adjustment for one recorder may or may not be proper for another. The adjustment routine is simple and it only has to be done once for a particular tape recorder. Here are the steps:

- 1) Insert a tape which already has data or programs on it. If you follow these directions the adjustment process will not harm your tape.
- 2) Plug two jacks into the playback plugs of the Expander 1. Plug the jacks at the opposite end into the remote and ear plugs of the recorder.
- 3) Set the tone control to slightly above midrange on your recorder.
- 4) Initially set the volume control at its lowest level on your recorder.
- 5) Turn the recorder on PLAY.
- 6) Wait five or ten seconds, and then slowly increase the volume on the recorder, until the small red light on the Expander 1 begins to glow. At this point, increase the volume a bit further (about 1/10 of the full scale).
- 7) This volume setting is the proper one for loading data from tapes. Make a pen or pencil mark on the recorder so you will be able to remember the volume position.
- 8) Stop your recorder and rewind your tape.
- 9) Your recorder is now adjusted for loading programs from tape to computer workspace. You will not have to do this again as long as you use the same recorder and remember to set the volume and tone controls at the proper level.

If you have any problems with the adjustment procedure, see your Expander 1 manual for help.

Now that your recorder has been properly adjusted, here are the steps for LOADING programs and data from the tape to computer memory:

- 1) Make sure the VideoBrain, Expander 1, and tape recorder are attached exactly as in the adjustment procedure above.
- 2) Set the tone and volume controls on the recorder to their proper position which you found during the adjustment procedure.
- 3) Rewind the tape completely.
- 4) Type LOAD on the keyboard but do **not** press (NEXT).
- 5) Put the recorder on PLAY and immediately press (NEXT) on the keyboard. The cursor will disappear from the screen.
- 6) When the data and programs are completely loaded, the words TAPE OK will appear on the screen along with the cursor.

- 7) Stop the recorder and rewind the tape for future use.
- 8) Type LIST to make sure that all you expected to get from the tape is now in the VideoBrain workspace.
- 9) If you received a TAPE ERROR message, you should redo this entire procedure. Check your Expander 1 manual for proper hookup and adjustment procedures.

You are ready to start over again. Notice that all the material in your previous workspace has been removed, and that only the information from the tape is in your workspace.

Again, if you have any trouble, check your Expander 1 manual for troubleshooting hints.

-4 VERIfy

Once you have saved the contents of your memory workspace onto tape, you should verify that the tape has indeed been recorded properly. If for some reason the tape is improperly recorded, you will not be able to use it for future LOADING. For this reason as soon as you have completed the SAVE operation, use the VERIfy command to make sure that your workspace is properly saved on tape. Follow these simple steps to VERIfy your SAVED tape:

- 1) Connect the jacks exactly as you would for the LOAD procedure.
- 2) Set the volume and tone controls exactly as you would for the LOAD procedure.
- 3) Rewind the tape which you just used to SAVE your present memory workspace.
- 4) Type VERIfy on the keyboard, but do not press <NEXT>.
- 5) Put the recorder on play and then immediately press <NEXT>. The cursor will disappear from the screen until the VERIfy process is complete.
- 6) When the tape has been VERIfied, a TAPE OK message will appear on the screen if the tape is good. You are now sure that you have a good copy of your workspace.
- 7) If a TAPE ERR message appears on the screen, your tape is **not** a good copy of your workspace. If this happens, redo the SAVE procedure in order to make a good copy. Then redo the VERIfy procedure in order to check this new copy of your workspace.
- 8) Stop your recorder and rewind the tape for future use.

Once you have successfully VERIfied the tape as being a good copy of your present memory workspace, you may erase any or all of your programs and data in memory. To reuse these lost programs and data, simply LOAD the tape that you have just SAVED and VERIfied.

-5 LOAD and SAVE example

Here is an example of the entire process. First, we will create and define some data. Then we will save the workspace. Then, we'll LOAD the material from tape and see that it's intact.

EDIT LODR	Puts you in edit mode
PROGRAM LODR	Sample program
"INPUT ROW DATA"	
DATA = KEYB	Input data from keyboard
BARColor 1 3 5 7 2 4 6	Sets bar colors
BARHeight DATA	Shows input data
ENDProgram	Ends program LODR
ENDEdit	Puts you in immediate mode
LODR	Runs program from immediate mode
MORE = 3 4 5 6	A new row variable
BARHeight MORE	Resets bars
LIST	Show programs & variables in workspace
SAVE	Tapes memory contents
VERIfy	Checks tape
⟨MASTER CONTROL⟩	Clears everything
LIST	Nothing in current workspace
BARHeight MORE	No variable "MORE"
LOAD	Input workspace from tape
LIST	Lists the whole workspace now
BARHeight MORE	Now "MORE" can be shown
LODR	Runs the LODR program

After pushing ⟨MASTER CONTROL⟩, the entire workspace is cleared. When you LIST, there is nothing there to see. The commands after that are for undefined variables — since the computer doesn't know them. Only when you reLOAD the values in your workspace do the variables MORE and the program LODR work.

Chapter 6: A Library of Sample Programs

-1 The Piracy Approach to Programming

A very good way to learn to program is to copy a lot of computer programs, try to understand them, and to modify them to suit your needs. This does not mean that you are not original, or that you are uncreative. It is simply the way we all learn to program.

The problem is simply to become a good pirate. By that we mean to learn to understand programs that you copy — or at least to be able to use them for whatever purpose you want.

While you are learning APL/S, copy as many programs as you can from this book onto tape and into your workspace. Change them. Modify them to do whatever you want done. When you are finished, you will understand the language and its capabilities much better than if you started every program afresh, and had to pound your way through each and every SYNTAX error.

To give you a good headstart in your piracy, we have included in this book a whole library of beginning programs which you might want to use or modify. Some of the programs are games; some are utility routines which you'll need in many other programs; some of the programs are oriented to business or finance. Among them, you should find some which will be interesting for you.

Try the programs, become familiar with the structure and commands which do things you're interested in doing. As you become more familiar with the language and its commands, your power as a programmer will increase dramatically.

-2 Using a Utility Program

A statement within a program may be the name of a program. When these statements are encountered, the flow of execution transfers to the first statement of the named program. At the end of that program, flow of control returns to the statement following the statement with the program name. In computer jargon, the subprogram is a subroutine and the statement with the subroutine name effects a "subroutine call". Because subprograms are often designed to be used with many other programs, the subprograms are frequently called utility programs.

A utility program is designed to perform a function — in a way similar to a system function. The system function LOG will give you the LOGarithm of a number to the base e. But there is unfortunately no system function to round. A good utility program would be one which you could use like this: `ROUND 23.48`. On page 51 is a program which does exactly that — except that you can't simply type `ROUND 23.48`. You have to give the program a variable which it can use, and the program will round that variable. Then the program gives you back the variable it rounded. Following is the full example.

PROBLEM: You want to round a number, using the simple routine `ROUND`. How do you do it?

THINKING IT THROUGH: The only variable rounded is a variable named `RNUMBER` (for Round `NUMBER`). So, the steps are

1. Give `RNUM` the value you want rounded
2. Let `ROUND` round `RNUM`
3. Take the rounded value back from `RNUM`, which is now rounded.

To see what happens when you use these steps, type in the following program and see how it works. Make sure that you have already typed in the program `ROUND` (on page 51) in edit mode before you try this example.

<code>DECIMAL = 3.847</code>	Your decimal number is 3.847
<code>RNUM = DECIMAL</code>	Give the decimal value to <code>RNUM</code>
<code>ROUND</code>	Calls the rounding subroutine
<code>DECIMAL = RNUM</code>	Gives rounded value back to <code>DECIMAL</code>
<code>DECIMAL</code>	Shows the new value of <code>DECIMAL</code> , which is 4

Try this with different values of `DECIMAL`. Notice that the program rounds whatever value you give it. If you want to see what happens during execution, type the word `ONTRACE` before you begin typing the example. It will trace every step of your work, as well as all steps of the round program. When you don't need to trace any more, type `OFFTRACE`, and the computer will stop tracing your every move.

-3 Designing Utility Programs

Utility programs should be designed with the knowledge that they will be used by other programs, so they shouldn't use names that you would otherwise want to use in the programs. They should be relatively short, and perform a single function. If they do more than one thing, they they become specialized, and you can't use them in a wide variety of situations.

PROBLEM: You would like a program which gives you the decimal part of a number. If you give it 48.3044, you want only the .3044 part returned, and the 48 chopped off.

THINKING IT THROUGH: What we would like to do, really, is ask the computer to look at the number and cut off the part before the decimal point. But the computer doesn't have eyes, so that won't work. One thing which does give us an idea, though, is this:

$$\begin{array}{r} 48.3044 \\ -48.0000 \\ \hline .3044 \end{array}$$

Subtracting the whole number part of the value will give us the decimal we want. Here, then, are the steps:

1. Find out what the sign of the value is.
2. Find the whole number part of the value.
3. Subtract it from the full value.
4. Multiply what is left by the sign of the original.

This basic series of steps results in the program listed below. Notice how each of the steps in Thinking It Through is included in the program — although they are all part of one logical line of text.

```
EDIT DECImal
XDEC = SIGN (XDEC)x          Don't type <NEXT> after this line
(ABS XDEC - (FLOOR(ABS XDEC))
ENDProgram
```

To use the program, here is a little example with a WHILE loop in it, so you can see how it works many times.

PROGRAM TRYDeci	Tries decimal program out
INNO = KEYB	Gives INNO value from the KEYBoard
WHILE INNO LT 1000	Loop stops when a value GT 1000 is put in
DO XDEC = INNO	Gives XDEC value of INNO
DECI	Calls DECImal program
INNO = XDEC	INNO gets decimal part back
INNO	Shows new value of INNO
"TYPE A DECIMAL "	
INNO = KEYB	Allows new value to be input
ENDWhile	Ends WHILE loop
ENDProgram	Finishes program

Notice that the structure of using APL/S utility programs is always the same. Find the name of the variable in the utility program which is changed. Then,

- 1) Give that variable the value that you want to be changed
- 2) Call the program
- 3) Take the value back from the variable the program changed.

With this in mind, here is a smorgasbord of programs for you to use. The first group are small, numerical routines you will no doubt find handy. The next group are just amusing, interesting programs you'll enjoy. The third group are some games we knew you would be able to program, understand, and have fun with.

-4 Utility Programs

ROUND

This program takes a variable called XROUND, and rounds it to the nearest whole number.

```
PROGRAM ROUND
RNUM = (SIGN RNUM)xFLOOR(.5+ABS RNUM)
ENDProgram
```

Chapter 6: A Library of Sample Programs

A more complete program, which lets you decide what place value to round to, is included below. Suppose you wanted to round an answer to the nearest 100, just give PV (Place Value) the value 100 before calling ROUND, and the number RNUM will be rounded to the nearest 100. If PV has the value 0.1, then RNUM will be rounded to the nearest tenth. For some programs, this ability will be most helpful.

```
PROGRAM PVRound
RNUM = (SIGN RNUM)xFLOOR((ABS RNUM)÷PV+.5)×PV
ENDProgram

INTeger
```

A program on page xx gives you the decimal part of a number. Here is a short program which gives the whole number (or integer) part of a number — chopping off the decimal.

The variable which gets changed is called NUMBER.

```
PROGRAM INTEger
NUMBER = SIGN (NUMBER) x FLOOR (ABS NUMBER)
ENDProgram

PAUSE
```

If you want a program to slow down so things don't happen in a blurry hurry, stick a few PAUSEs in it. Each PAUSE slows the program down a few seconds.

```
PROGRAM PAUSE
Q = 0
WHILE Q LT 10      Loop has Q count to 10
DO Q = Q + 1
ENDWhile
ENDProgram
```

-5 SCALE

Since the BARHeight function only shows numbers as large as 64, there are many situations in which you need to rescale the values of an array. What you want is the biggest possible scaling which will fit on the screen, while keeping all of the values in the array in proportion. The following routine rescales all values in the array BIGNumber correctly.

```
PROGRAM SCALE
SCLR = 64 ÷ MAX/ ABS BIGN
BIGN = BIGN x SCLR
ENDProgram
```

-6 Inverse Trig Functions

The ARCTangent, ARCSine and ARCCosine are not standard computer functions in APL/S. If you have need to compute them, the following set of programs will allow you to do so. The following polynomial series are used to approximate the value of the arctangent function ($\tan^{-1}$):

$$\tan^{-1} x = \frac{\pi}{4} \quad \text{for } x = 1$$

$$= -\frac{\pi}{4} \quad \text{for } x = -1$$

$$= x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \dots \quad \text{for } x^2 < 1$$

$$= \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \dots \quad \text{for } x > 1$$

$$= -\frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \dots \quad \text{for } x < -1$$

where x is in radians.

The ARCSine and ARCCosine programs are dependent on the ARCTangent function and used it as a subprogram. Therefore, ARCTangent must be in your workspace in order to run the other two.

Program ARCTangent

"INPUT VALUE"

V=KEYBoard

IF VxV EQ 1

THEN ANGLE = (SIGN V) x $\pi \div 4$

ELSE VI=(INDEX 5) x 2 - 1

VS=(INDEX 5) MOD 2 x 2 - 1

IF VxV LT 1

THEN ANGLE=+/(V*VI÷VIxVS)

ELSE ANGLE = (SIGN V)x $\pi \div 2 +$
 $(+ / (-VS \div (V * VI x VI)))$

ENDIf

ENDIf

"RADIAN ANGLE"

ANGLE

ENDProgram

Input value

Is V equal to 1 or -1?

Gives 1 3 5 7 9

Gives 1 -1 1 -1 1

Is V² less than 1?

Computes value of each term and
 adds them together

Displays radian value of angle

The above program uses five terms to compute the sum of the series. You may use more terms if you wish your answer to be more accurate. For example if you wished to use ten terms, lines six and seven should be changed so that they contain INDEX 10 instead of INDEX 5.

Chapter 6: A Library of Sample Programs

The following two programs, ARCSine and ARCCosine, use PROGRAM ARCTangent as a subprogram. In order to use them you must delete the first two statements after the line PROGRAM ARCTangent and the last two statements before the line ENDProgram, of the ARCTangent program. These four statements are designed so that the program ARCTangent can communicate with the user. If ARCTangent is to be used as a subprogram, it will not need to have user input or output.

The programs ARCSine and ARCCosine are based on the following two formulae:

$$\arcsin V = \arctan (V/\sqrt{1-V^2})$$

$$\arccos V = (\pi/2) - \arctan (V/\sqrt{1-V^2})$$

PROGRAM ARCSine	Main program
"INPUT VALUE"	
V=KEYBoard	
V=V÷((1-V*V)*0.5)	Computes new value
ARCTangent	Calls ARCTangent as a procedure
"RADIAN ANGLE"	
ANGLE	Displays value in radians
ENDProgram	
EDIT ARCCosine	Main arccosine program
"INPUT VALUE"	
V=KEYBoard	Input value
V=V÷((1-V*V)*0.5)	Compute new value
ARCTangent	Call ARCTangent as a procedure
ANGLE=PI÷2 - ANGLE	
"RADIAN ANGLE"	
ANGLE	Display ANGLE result
ENDProgram	

-7 COLR

With COLoR in your workspace, you never have to refer to a table of numbers to make a bar the right color. Just use the name and the color will show. First, put color in your workspace, then do the example at the bottom of the page to see how it works.

```
PROgram COLR
```

```
BLAK = 0
```

```
BLUE = 1
```

```
GREN = 2
```

```
AQUA = 3
```

```
RED = 4
```

```
PURPle = 5
```

```
BROWn = 6
```

```
SILVer = 7
```

```
GREY = 8
```

```
VIOLet = 9
```

```
LGREen = 10
```

```
LBLUe = 11
```

```
ORANge = 12
```

```
LPURple = 13
```

```
YELO = 14
```

```
WHITe = 15
```

```
ENDProgram
```

To use this program, type it into your workspace in edit mode, as you would any regular program. Then, in immediate mode, try the following commands:

```
COLR
```

```
BARHeight 10 20 30 40 50
```

```
BARColor BLUE,WHIT,RED,ORAN,LBLUe
```

```
BARColor BROW,AQUA,VIOl,YELO,GREY
```

```
BARColor LGRE,LGRE
```

Experiment with the different BARColors, and pick out combinations you like. You must use commas (the join operator) to separate color names.

```
SHIFt
```

This program rotates the values in a row variable, so that the first value becomes the last, and every other value shifts one place to the left.

```
PROgram SHIFt
```

```
N = 1
```

```
FIRSt = ROWV(1)
```

```
WHILe N LT SIZE(ROWV)
```

```
DO ROWV(N) = ROWV(N+1)
```

```
N = N + 1
```

```
ENDWhile
```

```
ROWV(N) = FIRSt
```

```
ENDProgram
```

Chapter 6: A Library of Sample Programs

After you've typed it into your workspace, use it in the small program below, and you can see how the program works.

PROGRAM TOPP	
COLOrs = 1 14 2 9 4 15 6	
BARS = 60 40 20 ~30 ~60 10 40	
WHILe 7 GT 4	Keeps going until you stop it
DO BARColor COLOrs	
BARHeight BARS	Show BARS with COLOrs
ROWV = BARS	
SHIFt	Moves bar heights 1 place left
BARS = ROWV	
ROWV = COLOrs	
SHIFt	Moves colors 1 place left
COLOrs = ROWV	
ENDWhile	
ENDProgram	End of program TOPP
ENDEdit	Puts in immediate mode
TOPP	

When your top stops spinning the colors and bars, you can see that the 7 values of BARS and COLOrs are always rotating to the left. This can be very useful, not only with bar colors, but also with many other row variables.

FMAX

This program finds the marker of the largest value in the row variable RMAX. While that doesn't sound too important, there are many situations in which it's necessary. For example, suppose you have a program which finds the maximum earnings of any of your stocks. That's great — but until you can find out which stock it is that's doing so well, knowing the maximum doesn't help you too much. In that situation, FMAX is a great help.

FMAX takes a row variable called RMAX, and returns the subscript (PTR) of the first maximum value in a variable called MAXI. Notice that if two values in the row variable are equal, FMAX will only find the first one.

```
PROGRAM FMAX  
PTR = 1  
WHILe RMAX(PTR) NE MAX/RMAX  
DO PTR = PTR + 1  
ENDWhile  
ENDProgram
```

SAFEty

This program makes sure that the value SAFX is greater than or equal to SMIN (the minimum legal number), and that SAFX is less than or equal to the maximum legal number, called SMAX.

```

PROGRAM SAFETY
WHILE (SAFX GT SMAX) OR
(SAFX LT SMIN)
DO "ILLEGAL VALUE OF"
  SAFX
  "INPUT NEW VALUE"
  SAFX = KEYB
ENDWHILE
ENDPROGRAM

```

-8 Amusing Programs

These programs are a bit frivolous. They are just interesting and fun. You'll probably want to keep them on a tape library to use with your friends.

SINE

This program traces a sine function as long as you want it to. It comes out as a wave on the bar graphs. The program shows negative sine values as red bars, and shows the positive ones as blue bars.

PROGRAM SINE	
N = INDE 7 ÷ 2	Gives N values .5, 1.0, 1.5, . . . 2.5
WHILE 1	Program stops only with (RUN/STOP)
DO B = SIN N × 60	Rescales N so it shows up
C = SIGN B × $^{-1}$ + 3	Sets colors
BARC C	Shows colors
BARH B	Shows bars
N = N + 0.5	Makes N .5 bigger
IF N (7) EQ 8	If N gets too big, reset it
THEN N = INDE 7 ÷ 2	
ENDIF	
ENDWHILE	
ENDPROGRAM	

SHRPshooter

This game selects two random numbers: a target number and a secret constant. You are told the target number, and your job is to intelligently guess the number, which when multiplied by the secret constant, will produce the target number. The target is between 1 and 10,000 and the secret constant is between 1 and 100. After you make your first guess the screen will show a white bar which is the target and another bar which is your shot. If your shot is too big, the second bar will be green. If your shot is too short the second bar will be red. If you hit the target exactly, the bars will flash on and off.

PROGram SHRPshooter

C =RAND 100	Secret constant
T =(RAND 100) x C	Target number
SC = T ÷32	SC bar scaling factor
R =0	R is result of C times your shot
WHILE R NE T	Play until you win
DO "YOUR TARGET IS"	
T	Display value of target
"TAKE A SHOT"	
SH =KEYBoard	Make a guess
R = SH x C	
CL =SIGN (T - R)+3	CL is color of second bar
H = T , R	H is height of bars
H = H ÷ SC	Scales height of bars
BARC 15, CL	
BARH H	
ENDWhile	
I =1	
WHILE I LE 5	Flashes bars on and off 5 times
DO BARH H	
BARH 0 0	
"DIRECT HIT"	
I = I +1	
ENDWhile	
ENDP	

GNUM

This game, GUESS A NUMBER, is a simple guessing game. The computer picks a random number between 1 and 100 and then asks you to make a guess. It will tell you if your guess is too large, too small, or exactly correct. If you make a correct guess, the computer will tell you how many guesses you made and the game will end. There is a best guess strategy that produces a win in the minimum number of guesses. Play the game and discover the best strategy.

EDIT GNUMber	
N=RAND 100	Target number
WINR=0	
NG=1	NG: number of guesses
WHILE WINR EQ 0	Play until WINR is not zero
DO "MAKE A GUESS"	
GUESS=KEYBoard	Enter guess
IF GUES EQ N	Is guess correct?
THEN "YOU WIN"	
"NMBR OF GUESSES"	
NG	Display number of guesses made
WINR=1	There is a winner
ELSE	
IF GUES LT N	Guess too small or too big?
THEN "TOO SMALL"	
ELSE "TOO BIG"	
ENDIf	
ENDIf	
NG=NG+1	Number of guesses increased
ENDWhile	
ENDProgram	

Chapter 7: Debugging

In computer lingo, bugs are the errors in a program. Debugging is the process of correcting these errors. If you have gotten this far in this book, you no doubt have had errors or bugs in your programs. This chapter will help you find and correct the bugs in your programs.

The APL/S language is designed to make debugging easy. Special functions and messages have been included in the language so that the computer itself can help you find your mistakes.

-1 Avoiding Bugs in Your Programs

The first rule of debugging is to avoid bugs by using good programming practices. A carefully written program will have fewer bugs than a poorly written one. Computer programming is one endeavor in which "an ounce of prevention is worth a pound of cure". Here is a list of good programming practices which will save you both time and trouble:

- 1) Think through your program step by step as if you were a high-level English-language computer. Write out each step in plain English. Use subprograms for the more complex steps. This will keep the top level program short and easy to understand. Refine each step until it is a legal APL/S statement. Repeat this process for each subprogram.
- 2) Write out your program neatly and clearly on a sheet of paper before you type it into the computer. Use indentation to help show the structure of the program. This allows you to look at the whole program at once.
- 3) Alongside your written program, you should make comments that will help you remember what a variable name stands for, or what a procedure does. This is called documenting your program.

If you follow these three suggestions, you will have fewer program bugs to worry about.

-2 Four Types of Error Messages

Computers are exacting machines. They require that you do things in a precise way. They tolerate absolutely no deviation. If you do something that is against the computer's rules of operation you will get an error message. APL/S has four types of error messages; no doubt you have already been confronted with some or all of them. What follows is a brief description of each type. Later on we will discuss these in more detail.

VALUE error: The computer does not know or will not accept the value your program has used in a computation.

Example: HITE = BIG will give you a VALUE 1 error message, if BIG has been assigned no value. HITE = 24 is legal.

SYNTAX error: The line you typed into VideoBrain does not follow the rules of proper APL/S grammar.

Example: `BARH = 2 4 6` gives SYNTAX 13 error message, because you cannot use `=` after `BARH`. `BARH 2 4 6` is the correct way to make the command.

CAPACITY error: You have too many numbers, commands, or variables in the computer at one time. It has run out of space to store all the information.

Example: If your program has too many lines for the computer to store in memory, you will get a CAPACITY error.

TAPE ERROR: Somehow, an error has occurred in either reading information from or writing information to a tape. Start the tape job over again. Tape errors are described in Chapter 5 on how to LOAD and SAVE information on tape.

A complete list of the VALUE, SYNTAX, and CAPACITY errors can be found in Appendix C. When you get an error, simply refer to this list to identify the type of problem.

-3 Immediate Mode Errors

Immediate mode errors are the easiest to find and fix. Every time you type a line and press `<NEXT>`, the computer checks to see if your command contained any errors. If it does, you will get an error message. If it doesn't the computer will execute your command as typed.

If you get an error message in immediate mode, you simply need to identify the error and then retype the line correctly. For example if you wish to assign the variable `TAX` the value 100, and you type

TAX 100

you will get an error message, "SYNTAX 6", and you will get your incorrect line displayed on the screen so that it may be retyped. Look in Appendix C to find the meaning of syntax error #6. Now look at the line you typed. You should have typed

TAX = 100

Backspace along the displayed line, retype it correctly, and press `<NEXT>`. This time the computer did not respond with an error message, which means that your line is OK. If you prefer to abandon your line, rather than correct it, you should key `<ERASE>`.

You may also get VALUE and CAPACITY errors in immediate mode. For some of these errors, the computer will display your incorrect line so that it may be retyped (as in the previous example). For other errors, the computer will not do this. In either case, you should look up the error type and number, decide why the computer rejected your command, and make the necessary changes. If the computer accepts the new version of your command, you have corrected the mistake successfully.

-4 Edit Mode Errors

Edit mode is used to create new programs and revise old ones. Almost all of the errors you will encounter in edit mode are of the SYNTAX type. Every time you type in a new line of your program and press <NEXT>, the VideoBrain checks to see if your line is grammatically correct, that is, if its syntax is correct. If the line is OK, then it allows you to type the next line of your program. If the line has incorrect syntax, the computer will give you an error message and in most cases will display the line again so that you may correct your error. The computer will not let you continue with your program until you have made the correction or erased the line using the <ERASE> key.

While in edit mode you will not receive any VALUE error messages, even though your program may contain VALUE errors. In edit mode, the computer does not check for this type of error. This error will be checked when you run your program. This topic will be discussed later in this chapter.

While in edit mode, you may receive a CAPACITY error if your program is too large, or there are already too many other programs or variables stored in memory, or if you attempt to enter a number which is too large or small. In order to avoid capacity errors that result from a shortage of computer memory, you should always remove useless variables or programs from memory by using the ERASE command. This will free extra memory space for new variables and programs. Be aware that once you use the ERASE command, you will no longer be able to recall the program or variable which you erased from your workspace. It no longer exists. (If you have the Expander 1, you may wish to store your programs on tape before you ERASE them.) Also remember that the ERASE command you type in is very different from using the <ERASE> key on the keyboard (see Chapter 1).

-5 Error Messages Upon Leaving Edit Mode

After you have typed in your program in edit mode, and then typed ENDEdit to return to immediate mode, the computer checks your program for SYNTAX errors. If it finds a syntax error, it will identify the error number and then display the line of the program that contains the mistake.

For example, look at the following program:

PRO gram COUN t	Program counts to 10
N =1	N starts at 1
WHIL e N LE 10	
N	
N = N +1	Print value of N
END While	End of WHILe loop
END Program	

Notice that the fourth line of the program is incorrect. This line should say, "DO N". At the time this incorrect line is typed, VideoBrain does not recognize it as an error. But when the programmer types ENDEdit, after finishing the program, VideoBrain will respond,

```
SYNTAX 101
N
```

SYNTAX 101 says that a line following WHILE is missing the DO keyword which must begin it. In order to correct this mistake, you must re-enter edit mode by typing, "EDIT COUNT", retype the line, and leave edit mode by retyping ENDEdit. At this point, if you have corrected the error, VideoBrain will accept your program without complaint.

Generally, the types of errors you will get upon leaving edit mode are syntax errors that refer to missing or incorrectly placed keywords in your program. Such words as DO, THEN, ENDWhile, ENDIf, and ENDProgram are likely to be misplaced or forgotten. Remember, in order to correct your error, you must re-enter edit mode, make the correction, and finally return to immediate mode to run the program.

-6 Run-Time Errors

Once a program is syntactically correct, that does not mean that it is completely correct. So far, you have been debugging your programs without actually running them. Often it is the case that errors which were not found during the editing process will turn up while the program is being executed by VideoBrain. These are called run-time errors.

Run-time errors generally are VALUE or CAPACITY types. Here is an example:

```
PROGRAM TEST
IF GAIN GT 0
THEN "DOING GREAT"
ELSE "DOING TERRIBLE"
ENDIF
ENDProgram
```

Program tests to see whether
gain is positive or not

This program will show no errors when it is being typed into VideoBrain using edit mode. But when it is run, it will produce a VALUE 1 error. This is a run-time error. When the computer started to execute the program it came to the line "IF GAIN GT 0", and searched its memory for the value of the variable "GAIN". It found no value had been assigned to GAIN. So how could it tell if GAIN was greater than or less than zero? It couldn't perform the test, so it stopped execution of the program and gave an error message, followed by the line that it couldn't understand.

Run-time errors immediately halt execution of the program. This is because VideoBrain doesn't know what to do, and rather than go on, it stops and tells the user, "I can't figure out what to do from here." That's when you have some debugging to do.

To correct a run-time error, you must re-enter edit mode, correct the error, and return to immediate mode to run the program. In the above example, PROGRAM TEST, you can correct the error by inserting two lines which allow the user to input a value for GAIN before reaching the IF statement. The two lines are:

```
"HOW MUCH GAIN?"
GAIN = KEYBoard
```

If you have trouble inserting these lines before the IF statement, see Chapter 8 for an example of how to insert.

You should be aware that just because you have found and corrected a run-time error, it does not mean that on your second attempt to run your program you will not find another error further down in your program. If you do, it, too, will have to be fixed.

What follows are some examples of the most common run-time errors. Key these examples into the computer and experiment with them.

ERROR: failure to give control variable a value before starting the WHILE loop.

```

PROGRAM ERR1
WHILE ZIP LE 100
DO ZIP  $\times$  2
    ZIP=ZIP+1
ENDWhile
ZIP
ENDProgram

```

Program ERR1 produces VALUE error #1 at the line, "WHILE ZIP LE 100". ZIP must have an initial value before the WHILE loop is entered. To correct it, insert

```
ZIP = 1
```

before the first line of the program.

Look at the next program, and see if you can figure out what's wrong with it before you get the error statement.

ERROR: two row variables of different size are used in an operation.

```

PROGRAM ERR2
COST = 1 2 3
ITEM = 1 2 3 4
COST  $\times$  ITEM
ENDProgram

```

Program ERR2 gives a VALUE error #3 when it gets to the line "COST $\times$ ITEM". The two row variables are not the same size, and therefore produce an error. To VideoBrain, it makes no sense to want to multiply 3 numbers by 4 numbers.

ERROR: A row variable subscript points to a number which is not in the row variable.

```

PROGRAM ERR3
FAMILY=10 20 30 40
P=1
WHILE P LE 8
DO FAMILY (P)
    P=P+1
ENDWhile
ENDProgram

```

This program produces VALUE error #6 at the line "DO FAMILY (P)". Each time the WHILE loop is executed, the value of (P) increases. When P equals 5, the statement FAMILY (P) will be evaluated to FAMILY (5). But there is no value for FAMILY (5). FAMILY only has four members. The pointer, (P), has gone out of bounds of the row variable. As a result, VideoBrain doesn't know what to do, and puts out an error message. To correct the error, limit the value of P to 4, so that it only points to one of the four members of FAMILY. Change the line beginning with WHILE to:

```
WHILE P LE 4
```

The following program bombs out on the last time through the WHILE loop. If you run it, you will quickly see why.

ERROR: An illegal value is used in a mathematical operation.

```
PROGram ERR4
BOMB= 10
WHILe BOMB GT 0
DO BOMB=BOMB-1
  1 ÷ BOMB
ENDWHILe
ENDProgram
```

Program ERR4 produces VALUE error #2 at the line "1 ÷ BOMB". It produces the error when BOMB has the value of 0, since it is illegal to divide by zero. Most of the mathematical operations have some value for which they are illegal. The bug in the above program can be fixed by exchanging lines 4 and 5:

```
DO 1 ÷ BOMB
BOMB=BOMB-1
```

-7 Infrequent Errors

Occasionally a programmer will write a program that he uses over and over again without finding any bugs, then on one particular occasion the program produces an error message. No changes have been made in the program. What could have happened? Look at this example:

ERROR: Input of an illegal value will stop the program.

```
PROGram MILLion
"INPUT YOUR NUMBER"
NUMBer=KEYBoard
"RESULT IS"
1000000 ÷ NUMBer
ENDProgram
```

This program will work perfectly until somebody keys the number zero in as a response to the keyboard prompt. At that time the program will fail. If NUMB**er** equals zero, then the line "1000000 ÷ NUMB**er**" is an illegal operation.

This type of infrequent error is fairly common in computer programs. Even though a program does not change, the data it operates on will change. How can this problem be solved? Use a trap in your program that will catch illegal values before they produce an error. An IF . . THEN . . ELSE . . ENDIf statement forms the body of the trap. Here is PROG**ram** MILL**ion** with the trap:

```
PROGram MILLion
"INPUT YOUR NUMBER"
NUMBer=KEYBoard
IF NUMB EQ 0
THEN "ZERO IS ILLEGAL"
ELSE "RESULT"
  1000000 ÷ NUMBer
ENDIF
ENDProgram
```

} This section traps illegal zero values.

-8 Infinite Loops

Up to this point, all of the examples of bugs produced an error message sooner or later. But there are programs which contain errors, though they do not produce error messages. One such program type is one with an infinite loop in it. Here is an example of an infinite loop:

ERROR: An infinite loop — one that will never stop “TWINKLing”.	PROGRAM STAR NEVER=1 WHILE NEVER LE 10 DO “TWINKLE TWINKLE” ENDWhile ENDProgram
--	--

Run program STAR. The program is clearly intended to show “TWINKLE TWINKLE” 10 times, but it keeps twinkling and twinkling without end. This is an infinite loop. The problem here is that the counter “NEVER” always stays equal to 1. It never changes, and so it is always Less than or Equal to 10. The programmer forgot to add one to the value of NEVER each time through the loop. If NEVER always has the value of 1, it will never be greater than 10, which is the condition for stopping execution of the WHILE loop.

This problem is easily fixed. Insert the line

NEVER=NEVER+1

directly under the line “TWINKLE TWINKLE”. Now this program will only twinkle 10 times, as intended.

In most infinite loop bugs, you will see nothing displayed on your screen because the loop does not contain a command to display words or numbers on the screen. In such a case, you may become impatient waiting for your program results. (You can always tell if the computer is executing your program, because the cursor disappears from the screen during execution.) Unless your program is doing computations with thousands of numbers, it is likely that if your program doesn’t finish in less than five minutes, you are stuck in the middle of an infinite WHILE loop.

You can stop your program by holding down the blue <STOP> key until the word STOPPED appears on your screen, along with the last line executed. This line will tell you in which WHILE loop you are stuck.

To fix your bug, you must remember that you are stuck in an infinite WHILE loop because the condition you established for finishing the loop is never met. In the previous “TWINKLE TWINKLE” example, the variable NEVER was not increased in value, and therefore it never satisfied the condition (being greater than 10) for completion of the loop.

Look at your program as it is written out on paper. Follow the WHILE loop through a few times as if you were the computer. Trace the value of the conditional variable. Will the condition ever be satisfied? If not, you have found your problem, and you can then correct it.

–9 Conceptual Errors

There will probably be occasions in your programming in which your program produces no error messages and doesn't get stuck in infinite loops, but the program is still in error because it produces incorrect results.

For example, if you write a program to figure your income tax and the results tell you to pay \$38,453 in taxes, but you know you only earned \$33,000 in income, the program has a bug. The results are not reasonable. You have obviously made a conceptual error in writing your program. Maybe instead of subtracting your deductions from your income, you mistakenly added them on. Finding conceptual errors and fixing them are the hardest tasks in debugging programs.

Here are some suggestions for finding and correcting conceptual bugs in your programs.

Rethink your program. Go back to the original version of your program that is written in simple English. Do you have the steps in proper order? Did you forget to include a step?

Check APL/S statements. Did you correctly translate the plain English description of your program into APL/S? Is the order of APL/S functions correct?

Check the computer version. Did you type your program into the computer exactly as you wrote it out on paper? You may have forgotten a line, or you may have typed + when you meant –.

Check final values of variables. After your program has completed execution, you should check to see what are the final values of all your variables. You can do this by simply typing, in immediate mode, the variable name and then pressing (NEXT). The value of that variable will appear on the screen. You may find that a variable is larger or smaller than expected. This would indicate that your bug may be in how this variable is handled in your program.

Trace the value of a variable. If you suspect a certain variable as the cause of the bug, you should follow the value of this variable through the whole program. What is its value at the beginning of the program? What is its value after it has completed a series of while loops? You may do this type of checking by hand (that is, pretending you are the computer) or you may wish to use ONTrace and OFFTrace, which are discussed in the next section.

Use immediate mode. You can check complex statements or procedures in immediate mode, seeing whether the function produces the expected values.

Ask a friend to help you. If you have a friend who knows how to program, ask her (or him) to help in correcting your bug. Even if your friend is not familiar with APL/S, he or she may be able to offer assistance. Often just explaining the problem gives you insight into it, whether or not the other person can actually help with the programming. Although this is the last suggestion in the list, it shouldn't necessarily be the last thing you try. You should talk your bug burden over with a friend before it gets too heavy. Don't be afraid to ask. Most computer people like to help solve problems. It's one thing which makes working with computers interesting.

-10 ONTRace, OFFTrace and <STOP>

When you are debugging your program, you will often find it necessary to actually follow the execution of your program step by step. You may do this by hand, which means that you execute the program using paper and pencil. Or you can use the ONTRace and OFFTrace commands.

The ONTRace command will show each line of a program which the computer is running. It will also show you the value of crucial variables as they are computed. In order to see how it works, make sure you have the corrected version of PROGRAM STAR (see page 67) in computer memory. Now type the following in immediate mode, and watch your screen:

ONTRace	Turns tracing system on
STAR	Runs PROGRAM STAR

Notice that each line of the program is displayed just before it is executed. Also notice that the value of each variable on the left side of the = sign is displayed directly under the variable name.

If the program went too fast for you to see, you can press the <STOP> key once to halt the program, look at the trace, then push <NEXT> to restart it. You can repeat this process as often as you wish until the program is completely finished running. But be careful not to hold the <STOP> key down too long. This will completely interrupt the running of your program, and the word STOPPED will appear on the screen. If this happens, you must type STAR once again to run PROGRAM STAR.

Once you have corrected your error, you will not need to use the trace system further. Type OFFTrace, and the system will return to normal execution. To try it, in immediate mode type:

OFFTrace	Turns trace system off
STAR	Runs program STAR normally

In case you only want to trace part of a program, you may insert the commands ONTRace and OFFTrace as part of the actual program at the time you enter it into memory. If you choose this option, the program will run as usual until it finds the ONTRace command. At this point the program will be traced on the screen, statement by statement, until the computer encounters OFFTrace, or until the program is finished. If a program is finished with ONTRace still in effect, the next program run will be traced, until an OFFTrace command is found in the program, or until OFFTrace is entered in immediate mode. You may insert as many pairs of ONTRace and OFFTrace commands in your program as you wish.

Chapter 8: Alphabetical Reference Section

This section provides an alphabetical list of commands, statements and functions. Commands manipulate programs, data and the workspace. Commands cannot be used inside a program. Statements can be used in the immediate mode or inside a program. Exceptions include the IF and WHILE statements, which can only appear in programs. Functions manipulate data and are used within statements to generate the desired results.

-1 Commands

EDIT

The EDIT command is used to create programs and to change programs. The command initiates the edit mode. The program to be edited or created is specified in the EDIT command by its *program-name*. The ENDE command ends the edit mode.

Format

EDIT *program-name*

Notes:

1. If *program-name* refers to a new program, then the statement “PROG *program-name*” is automatically created. A black cursor is positioned on the next line and program statement entry is enabled.
2. For new programs, the user must enter an ENDP statement as the last statement of a program.
3. In the edit mode, the blue keys on the VideoBrain keyboard have the following meanings:

- ⟨BACK⟩ The ⟨BACK⟩ key moves the cursor back one position and obliterates the previous character. The ⟨BACK⟩ key can be used to correct or change a statement.
- ⟨PREVIOUS⟩ The ⟨PREVIOUS⟩ key moves the cursor to the end of the previous statement.
- ⟨NEXT⟩ The ⟨NEXT⟩ key moves the cursor to the end of the next statement. If no statements follow the current statement, the cursor is positioned at the beginning of a blank line.
- ⟨SPECIAL⟩ The ⟨SPECIAL⟩ key positions the cursor at the beginning of a blank line. It can be used as follows:
 - 1) To insert a statement into a program.
 - 2) To allow entry of the ENDE command that takes the user out of the edit mode.

- ⟨ERASE⟩ The ⟨ERASE⟩ key deletes the current statement (an erase character ■ is positioned at the end of the old statement) and positions the cursor at the beginning of a blank line. The ⟨ERASE⟩ key can be used to erase a logical line (key ⟨NEXT⟩ following the ⟨ERASE⟩ key) or to replace a logical line.
4. In the edit mode, each statement is checked for syntax errors whenever the ⟨NEXT⟩, ⟨PREVIOUS⟩ or ⟨SPECIAL⟩ keys are pressed. If the current statement has an error, an error message is displayed and the current statement is displayed again. The cursor is positioned at the end of that statement to allow editing of that statement.
 5. See Chapter 7 for examples of editing sessions.
 6. When ENDE is entered the program is checked for proper syntax. For example each IF must have a corresponding THEN and ENDIF. If an error is found the statement which was current at the time of error detection is displayed, and the program is marked non-executable. In this case, the user should edit the program to correct the error before running the program.
 7. The PROG statement cannot be edited.

ERASe

ERASe removes a variable or program from the workspace.

Format

ERASe *name*

Where *name* refers to a variable or program.

Notes:

1. Erasure of a variable or program with references from remaining programs in the workspace could result in unpredictable states.
2. If the user lists a statement of a program which contains a reference to the erased variable or program, the name of the referenced program or variable will be changed. If the memory location has not yet been reused, then the first character of the name will display as a solid square. If the location has been reused, then the name of the new variable will be used.
3. The ERASe command is not the same as the ERASE editing key.

LIST

LIST generates the list of variable names and program names currently in use in the workspace.

Format

LIST

Notes:

1. The list of names is displayed unsorted.
2. LIST also displays the bytes of RAM storage available for program and data storage.

LOAD

The LOAD command loads the contents of a cassette into the workspace.

Format

LOAD

Notes:

1. Any existing data or program in the workspace will be replaced by the contents of the tape. If the content of the workspace is to be saved, use the SAVE command before starting the LOAD sequence.
2. The LOAD sequence proceeds as follows:
 - a. Make sure the Expander 1 and the tape recorders are attached properly. (See the Expander 1 manual for complete instructions).
 - b. Rewind the tape completely.
 - c. Put the recorder on PLAY.
 - d. Type LOAD followed by the <NEXT> key into the VideoBrain keyboard. The cursor will disappear until the contents of the tape are loaded into the workspace.
 - e. If proper load occurs, APL/S will display a "TAPE OK" message and redisplay the cursor.
 - f. When the cursor reappears, enter the LIST command to make sure that the data and programs are properly loaded.
3. If problems occur, check with the Expander 1 manual for troubleshooting hints.

SAVE

The SAVE command saves the contents of the current workspace on cassette tape.

Format
SAVE

Notes:

1. SAVE stores a copy of the current workspace on tape. The contents of the workspace is left intact after the SAVE command.
2. SAVE stores variable names, values and programs on tape.
3. The SAVE sequence proceeds as follows:
 - a. Make sure the Expander 1 and the tape recorders are attached properly. (See the Expander 1 manual for complete instructions).
 - b. Rewind the tape completely.
 - c. Set the recorder on RECORD.
 - d. Type SAVE followed by the <NEXT> key into the VideoBrain keyboard. The cursor will disappear until the contents of the workspace is stored on tape.
 - e. When the cursor reappears, remove the cassette tape and label appropriately for future use.
4. SAVE automatically records the data three times to assure proper recovery of the workspace.
5. Make sure the recorder is running during the SAVE operation. VideoBrain might display the TAPE OK message even if the recorder is not running (i.e., the recorder is not plugged to an AC outlet, the REMOTE jack does not mate correctly).

VERIfy

The VERIfy command checks that any copy of the workspace saved on tape is saved correctly and can be loaded. Unlike the LOAD command, VERIFY does not load the copy on tape into the workspace and does not alter the workspace in the computer in any way.

Format
VERIfy

Notes:

1. VERIfy does not compare the workspace with the workspace copy on tape. Thus you can verify a tape while holding an entirely different set of data and programs in the workspace.
2. VERIfy and LOAD both use an error checking code which will report a tape error if the data and programs on the tape are not stored correctly.
3. The VERIfy sequence proceeds as follows:
 - a. Make sure the Expander 1 and the tape recorders are attached properly. (See the Expander 1 manual for complete instructions).
 - b. Rewind the tape completely.
 - c. Put the recorder on PLAY.
 - d. Type VERIfy followed by the (NEXT) key on the VideoBrain keyboard. The cursor will disappear until the contents of the tape are verified.
 - e. If the tape is verified, APL/S will display a "TAPE OK" message and redisplay the cursor. Otherwise a "TAPE ERR" message will be displayed.

-2 Statements**Assignment**

Assignment statements save the result of a computation in a variable. The variable and its values are stored in the workspace.

Case	Format
1.	<i>variable=value</i>
2.	<i>variable(subscript)=value</i>

Examples:**Input**

```

I=0
ROW=10 20 30
T=KEYB÷100
X=ROW
P=2 LOG 7
G = INDEX 5
ROW(2) = 1
A=INDEX 10
I=4
A(I-1)=A(I)

```

I is assigned the value of 0.
 ROW is assigned the values of 10, 20, and 30.
 The result of keyboard input ÷ 100 is assigned to T.
 The values of ROW is assigned to X.
 The base 2 log of the value 7 is assigned to P.
 The integers 1 through 5 are assigned to G.
 The second element of ROW is assigned the value of 1.
 The integers 1 through 10 are assigned to A.
 I is assigned the value 4.
 The Ith value of A is assigned to the (I-1)th element of A.

Notes:

- 1. The variable and its value are stored in the workspace. The SAVE command stores these variables and values as well as any programs in the workspace on cassette tape. Subsequent LOAD commands retrieves these variables, values and programs. Thus the assign statement can be used to manage a database.
- 2. If *value* cannot be calculated, an ERROR results and the assignment is not performed.
- 3. In case 2 of the assign statement the variable must be an array, *subscript* must evaluate to a scalar (or one element array) positive integer from 1 to the size of the array. Also, *value* must evaluate to a scalar or one element array.
- 4. Multiple assignments may occur in a single statement. Legal examples appear below.

Input

```
I=J=0
D=20÷((R=1+I÷100)*30)
```

I and J are assigned the value 0.
R is assigned the value of 1+I÷100 and D is assigned the value of the entire expression.

Application: Managing a Stock Portfolio

Doctor Randolph Rucker has a portfolio of 5 stocks containing the following:

	Shares	Current Price	Beta
1. Intel	100	53.75	2.3
2. Ford	100	67.50	1.2
3. AT&T	75	87.00	1.0
4. IBM	50	220.50	1.3
5. Xerox	25	67.25	1.5

Doctor Rucker would like to create a data base containing this data, keep it up-to-date and periodically compute the value of his portfolio and the portfolio beta. The following assignment statements create his data base.

Input

```
SHS=100 100 75 50 25
PRC=53.75 67.50 87. 220.50
    67.25
BETA=2.3 1.2 1.0 1.3 1.5
```

Create an array variable SHS
Create an array variable PRC
containing the current price
Create an array variable BETA

Doctor Rucker sells 30 shares of his Intel stock. Also, the current Wall Street Journal lists the most recent closing price of his stocks as 69.375, 62.25, 88.50, 280.00 and 73.50 respectively. The following assign statements updates his database.

Input

```
SHS(1)=SHS(1)-30
PRC=69.375 62.25 88.50 280. 73.50
```

Reduce Intel shares by 30.
Update the current price variable
PRC.

To compute the value and the beta of his portfolio, Dr. Rucker can enter the following:

Input

```
+/(MV=PRC×SHS)      33556.25
+/(MV×BETA)÷+ /MV    1.27778
```

BARC

Usage: BARC assigns colors to bars of the bargraph.

Format

BARC *value*

Where *value* can be a scalar or array.

Examples:

Input

Result

```
BARC 1
BARC 0 1 2 3

BARC INDEX 7
BARC ARRAY 7+4
```

Color 1 is assigned to bar 1.
Colors 0, 1, 2, and 3 are assigned to bars 1, 2, 3, 4
respectively.
Colors 1 to 7 are assigned to the bars.
Set all bars to the red (4) color.

Notes:

1. *Value* is converted to a number from 0 to 15 using the (*value* MOD 16) formula. The numbers 0 to 15 have the following color meaning.

Code	Color	Code	Color
0	Black	8	Grey
1	Blue	9	Violet
2	Green	10	Light Green
3	Aqua	11	Light Blue
4	Red	12	Orange
5	Purple	13	Light Purple
6	Brown	14	Yellow
7	Silver	15	White

- 2. The graph background color is black (0). Therefore, any bar assigned a color of 0 will not be visible.
- 3. The initial color of all bars is red (4). Whenever <MASTER CONTROL> is used, the bar colors are all reset to red.
- 4. BARC uses only the first seven elements of *value*. Elements of *value* beyond the seventh are ignored. If less than seven elements are specified by *value* the colors of the unspecified bars are left at their previous color settings.

BARH

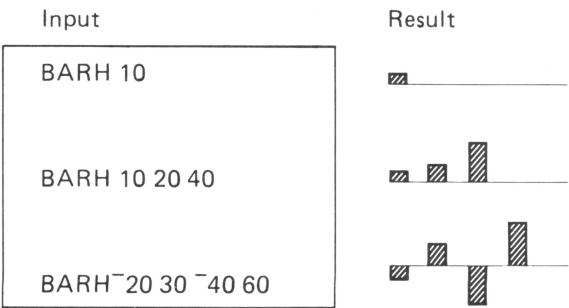
BARH controls the height of the bars on the graph area.

Format:

BARH *value*

Where *value* represents a scalar or array.

Examples:



Notes:

- 1. The bars have an absolute scale from -64 to 64. Heights greater than 64 are reduced to 64 and heights less than -64 are increased to -64.
- 2. BARH uses only the first seven elements of *value*. Elements beyond the seventh are ignored. If less than seven elements are specified, the heights of the unspecified bars are left at their previous heights.

Application: Logarithmic Plotting

A logarithmic plot preserves the proportion between the logarithms of two values. For example, the ratio of the logarithms of 10 and 100 is 1 to 2. Logarithmic plots are used to display values which may differ by very large amounts. The program below generates a logarithmic plot for an input array.

PROG LOGP	
IN=KEYB	Accept array from keyboard.
OUT=LOG IN	Convert to logarithmic values.
HIGH=MAX/OUT	Find largest value.
LOW=MIN/OUT	Find lowest value.
SCAL=64÷(HIGH-LOW)	Compute scaled unit height.
BARH (OUT-LOW)×SCAL	Compute bar heights and display.
ENDP	

Character-constant

A *character-constant* is any string of alphabetic or numeric characters. It can be used in programs to prompt for user inputs or to annotate computed answers.

Format:

"character-constant"

Examples:

Input	Result
"ENTER RATE?"	ENTER RATE?
"THE ANSWER IS:"	THE ANSWER IS:

Notes:

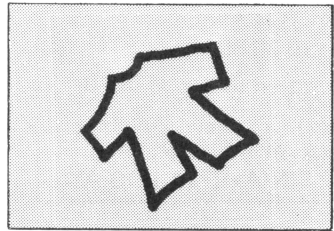
1. The quotes (<">) are used to delimit the *character-constant*. Therefore, the *character-constant* cannot have imbedded quotes.
2. When evaluated, the character-constant statement is displayed without the quote marks.

Application: Statistical Analysis of a
Clothing Budget

The Jones' collected some figures on their expenditures for clothing. The figures appear below. What is the average, variance and standard deviation of the Jones' monthly clothing expenditure?

	JAN	FEB	MAR	APR	MAY	JUN
Clothing	43	0	412	26	106	77

	JUL	AUG	SEP	OCT	NOV	DEC
Clothing	0	310	87	15	210	46



```

PROG STAT
"ENTER MONTHLY FIGURES"
IN=KEYB
N=SIZE IN
M=+/IN÷N
V=+/(INxIN)÷N-MxM
SD=V*.5
"MEAN, VAR., S.D."
M, V, SD
ENDP
  
```

Prompting for input.
Store array values in variable IN.
Determine size of array.
Compute mean.
Compute variance.
Compute standard deviation.
Labeling the output.
Display results.

Input/Result

STAT
ENTER MONTHLY FIGURES
743 0 412 26 106 77 0
■310 87 15 210 46
MEAN, VAR., S.D.
111 15974.33 126.3896

Run program.
Prompt.
Keyboard input

Output label.
The average monthly clothing expense is
\$111 with a standard deviation of \$126.39.

Display

The display statement shows the result of a computation

Format

value

Examples:

Input	Result
16	16
3÷2.7	1.111111
SIN π	0
A=10	
A	10
(A-1)x4	36
B=10 20 30	
B	10 20 30
B-20	-10 0 10
B(1)	10
B(A-9)	10
A, SIN(AxII),(A LOG A)	10 0 1

Notes:

1. The display statement is very similar to an assignment statement (i.e., any legal expression will be evaluated); however, instead of storing the results in a variable, the results are displayed.
2. The display statement may contain an imbedded assignment. The statement, FVx((R=1+I÷100)*N) will store the intermediate result, 1+I÷100 in the variable R and display the value of the entire expression.
3. If *value* cannot be calculated, an error results and no answer is displayed.

If is a control statement which allows conditional execution of APL/S instructions.

Case	Format	Use
1.	IF <i>condition</i> THEN <i>block</i> ENDI	If <i>condition</i> equals 1. Then <i>block</i> is executed.
2.	IF <i>condition</i> THEN <i>block1</i> ELSE <i>block2</i> ENDI	If <i>condition</i> equals 1. Then <i>block1</i> is executed. Else <i>block2</i> is executed.

Where *condition* is an expression that evaluates to a 1 or 0 value
and *block* represents a sequence of zero or more APL/S statements.

Notes:

1. The IF *condition* must evaluate to a 0 or 1 scalar or one-element array value. Otherwise, an error is generated.
2. The keywords IF and ENDI delimit the IF statement. The THEN keyword begins the block of statements that is executed if the *condition* is true. The optional ELSE keyword begins the block of statements that is executed if the *condition* is false. The IF, ENDI, THEN and ELSE keywords must each begin on a new logical line.
3. The IF statement can only be used inside a program.

Application: Quadratic Roots

The roots of a quadratic equation of the form $ax^2+bx+c=0$ can be solved by the following formulas:

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The following program computes the quadratic roots of any quadratic equation.

PROG QUAD

```
A=KEYB
B=KEYB
C=KEYB
I=BxB-4xAxC
IF I LT 0
THEN I=-I*.5÷(2xA)
      R=-B÷(2xA)
      "REAL +/-IMAG"
      R, I
ELSE I=I*.5÷(2xA)
      R=-B÷(2xA)
      "REAL ROOTS"
      (R+I),(R-I)
ENDI
ENDP
```

Input coefficients.

Compute the square root term.

If term is negative.

Compute imaginary part.

Compute real part.

Display answer.

Compute determinant.

Compute real part.

Display answers.

KEYBoard

(Input Function)

KEYBoard allows values to be entered from the keyboard during program execution.

Examples:

Input

```
ABS KEYB
+ /KEYB
KEYB x KEYB
IN=KEYB
J=KEYB MAX 1E70

NOP(3)=KEYB
```

Display the absolute value of input value.

Display the sum of any input value.

Multiply the first input value by a second input value.

Assign an input value to the variable IN.

Assign the larger of the input value of 1E70 to the variable J.

Assign the input value to the 3rd element of array NOP.

Notes:

1. KEYB displays a question mark to prompt for keyboard input.
2. The user response to the keyboard input request can only be an empty line, a scalar
3. If the response is an empty line, program execution is ended.

Application: Chained Addition

The following program performs chained addition and subtraction.

```

PROG CHAIN
R=0
WHILE 1
DO R=R++/KEYB
  R
ENDW
ENDP

```

Initialize R.
Set-up infinite loop.
Compute commulative sums.
Display new result.

Input/Result

```

CHAIN
?350.14
350.14
?^27.00
323.14
?^14 ^50 ^25
234.14
?(BACK)<(NEXT)
ENDED
PROG CHAI
DO R=R++/KEYB

```

Run program CHAIN.
Enter value.
Program response.
Enter negative number.

Enter array of numbers.

Enter an empty line to exit program.
Execution is ended.
In program CHAIN.
At this statement.
Now in immediate mode.

ONTRace/OFFTrace

ONTRace initiates program line tracing for program debugging. OFFTrace turns off program line tracing.

Format	Use
ONTR	Turn-on trace.
OFFT	Turn-off trace.

Notes:

1. ONTR and OFFT may be used as an immediate mode command or as a statement in a program.
2. When tracing is turned on, each program statement is displayed before it is executed. After a line is executed, any value generated in an assignment statement is automatically displayed.

The *program-name* statement is used to run a program when used in the immediate mode or to call another program when used inside a program.

Format

program-name

Notes:

1. Program execution is initiated by entering the *program-name*. Normal program termination occurs when the logical flow of the program encounters the ENDP statement inside the program.
2. To temporarily stop execution, key ⟨STOP⟩. A second ⟨STOP⟩ entry prematurely ends execution. A ⟨NEXT⟩ entry following the first ⟨STOP⟩ key allows the program to resume execution.
3. Recursive program calls are allowed.

Application: Reverse Values

The program, REV, reverses the values in the array IN and stores the results in the array, OUT. For example, if IN contains 10 20 30 40, then OUT would contain 40 30 20 10. The second program, BARS, uses the program REV to reverse the bar heights repeatedly, first showing an up-incline, then a down-incline, and so on.

```

PROG REV
N=SIZE IN
OUT=ARRAY N
I=1
WHILE I LE N
DO OUT(I)=IN(N-I+1)
  I=I+1
ENDW
ENDP

```

A program that reverses the values in the array IN.

```

PROG BARS
IN=INDEX 7x7-50
WHILE 1
DO BARH IN
  REV
  IN=OUT
ENDW
ENDP

```

A program that plays with the bar graphs.

Call program REV to reverse values in IN.

The session below runs both programs. Note that the program REV can be called from the immediate mode as well as from another program.

Input

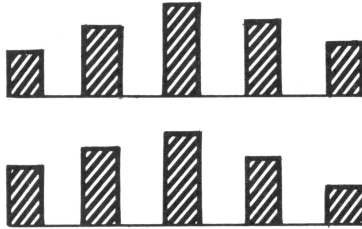
IN=20 40 60 50 40

BARH IN

REV

BARH OUT

BARS



WHILE

WHILE is a control statement which allows repeated execution of a block of APL/S statements.

Format

Use

WHILE *condition*

Block is repeatedly executed as long as *condition* has the value 1.

DO *block*

ENDW

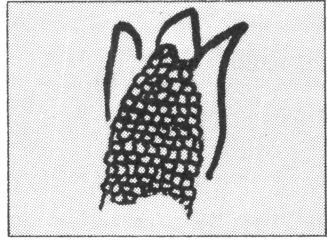
Where *condition* is an expression that evaluates to a 1 or 0 and *block* represents a sequence of zero or more statements.

Notes:

1. The WHILE *condition* must evaluate to a 1 or 0 scalar or one-element array value. Otherwise, an error is generated.
2. The keywords WHILE, and ENDW delimit the WHILE statement. The DO keyword indicates the beginning of the block of repeated statements. The WHILE, DO and ENDW keywords must each begin on a new logical line.
3. *Block* may contain the statement EXIT, which immediately terminates the execution of the WHILE block.
4. The WHILE statement can only be used inside a program.
5. The statement WHILE 1, causes infinite repetition of *block*. Execution can be terminated with the (STOP) key or EXIT statement.

Application: The Fibonacci Series

The Fibonacci series are numbers that characterize relationships found in mathematics and nature. For example, the grains of an ear of corn follows a series of Fibonacci numbers.



The series can be generated by summing the preceding two elements starting with the values 0 and 1. The first 7 elements would consist of 0, 1, 1, 2, 3, 5, 8 . . .

The quotient of successive terms in a Fibonacci series converge to a value often called the "golden ratio". This ratio was used by the ancient Greeks to design the most aesthetically appealing building (width to length ratio of buildings).

The program below generates the Fibonacci series and computes successive approximations of the "golden ratio".

PROG FS	
A=R=0	Initialize first value of Fibonacci series A and value of golden ratio R.
B=T=1	Initialize second value of series B and value of temporary storage T.
WHILE R NE T	Begin repetition statement.
DO C=A+B	Compute new value.
A=B	Reassign values for A
B=C	and B.
T=R	Store old value of R.
R=A÷B	Compute new golden ratio.
C,R	Display new value in series and new ratio.
ENDW	
ENDP	

-3 Functions

Introduction to Functions and Their Formats

Functions can work with either scalars or arrays. The result generated can be displayed, assigned, or used as input to another function. APL/S functions can be used in one of the following three formats: one-argument, two-argument and array reduction. Some functions have only one permissible format. Other functions perform different operations depending on the format used. This introduction discusses one and two-argument functions by type of format. The array reduction format is discussed in the reference section on the reduction operator.

One-Argument

One-argument functions, like sine or absolute value, require only one argument. When applied to a scalar, one-argument functions generally return a scalar as the result; when applied to an array, the result is generally an array of the same size. The one-argument functions that do not follow these general rules are the mixed-functions (ARRAY; INDEX; and SIZE).

The format for use of one-argument functions is:

function value

Where *value* can be either a scalar or an array. *Value* can also be the result of another function. A few examples using scalars follow:

Input	Result	
÷ 20 10	.05 0.1	Reciprocal
LOG 10	2.302585	Natural logarithm (Base e)
FLOOR 7.38	7	Largest included integer
SIN π	0	Sine of π
LOG ABS 1	0	Log of the absolute value of 1
!5	120	Factorial of 5

When arrays are used, the function is applied to each number in the array to produce a corresponding number. The result is an array of the same size as the input array.

Examples follow:

Input	Result
ROW=2 8 .25	no display
÷ROW	0.5 0.125 4
EXP ROW	7.38706 2980.957 1.284025

Mixed One-Argument Functions

In general, APL/S functions generate scalars when the inputs are scalars and arrays when any input is an array. The three functions, INDEX, SIZE, and ARRAY are the exceptions.

These functions generate arrays or transform arrays into scalar information. A brief description of their use follows:

INDEX generates an array with values from 1 to the indicated scalar value. For example:

Input	Result
INDEX 5	1 2 3 4 5 Create an index array

Chapter 8: Alphabetical Reference Section

ARRAY generates an array containing the specified number of elements. Each element is assigned the value of 8. For example:

Input	Result
ARRAY 3	0 0 0 Creates an array with three zeroes

SIZE returns the size (the number of elements) of an array. For example:

Input	Result
ROW=15 3 2 .5 SIZE ROW	4 Size of the array ROW

The concatenation function (,) is a mixed function which can take either one or two arguments.

Two-Argument

Two-argument functions, like addition (+) or modulo, require two arguments to generate a result. The arguments can be two scalars, two equal-sized arrays or a scalar and an array. The concatenation function (,) is the only function for which the arguments can be arrays of unequal size.

The format for use of two-argument functions is:

value function value

Where *value* can be a scalar or array.

When two scalars are used as inputs, the result is a scalar (except for concatenation). Some examples follow:

Input	Result	
8-5	3	Subtraction
3x9	27	Multiplication
52÷13	4	Division
3*2	9	Raising to a power (squaring)
10 LOG 100	2	Base (10) logarithm.
4 MIN π	3.141593	Minimum
4.3 LT 4.4	1	Less Than condition

When arrays and scalars are used together as inputs to a two-argument function, the result is an array of the same size as the input array. For example:

Input	Result
RIG=81 15 6 RIG÷3 2xRIG	27 5 2 162 30 12

When using two arrays as inputs, the two arrays must be the same size and the result is an array of the same size. The exception to the size rule is when one of the arrays is an array with a single element. Then, the single-element array is treated by APL/S as if it were a scalar.

When two arrays are used, the function is applied to each pair of numbers in the two arrays. An example follows:

Input	Result
VEC1=1 2 3 VEC2 = 10 20 30 VEC1xVEC2	10 40 90

A summary of the types of inputs to two-argument functions and the resulting types of output is given in the following table:

Two-argument Function Size Requirements

Input		Input		
Input	Scalar	Empty Array	Single Element Array	Array
	Scalar	Empty Array	Single Element	Array
	Empty Array	Empty Array	Empty Array	Error
	Single Element Array	Single Element	Empty Array	Single Element
	Array	Array	Error	Array



+ computes the sum of one or more terms.

Case	Format	Use
	$+ \textit{value}$	Additive identity
	$\textit{value} + \textit{value}$	Addition
	$+/\textit{array}$	Summation

Where *value* can be a scalar or array.

Input	Result
+7	7
2+3	5
(2 7 10)+5	7 12 15
(3 4)+(8 6)	11 10
+/(2 4 6 8 10)	30

Note:

The reduction, $+/\textit{array}$ sums all elements of an array. It is evaluated as follows:

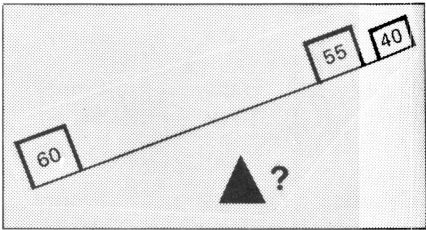
$$\textit{array}_1 + \textit{array}_2 + \dots + \textit{array}_n$$

If *array* is an empty-array, the result is a 0.

Application: Center of Mass

Bobby, Jennifer and Irene all want to ride the seesaw. Their weights and distance from the center of the board are as follows:

	Bobby	Jennifer	Irene
Weight	60	55	40
Distance	-5	4	5



What is the fulcrum of the seesaw that will allow all three to use the seesaw at the same time?

The point where the weights of the children are properly balanced is called the center of mass. The formula is:

$$\bar{x} = \frac{\sum x \cdot m}{\sum m}$$

The example below determines the center of mass for the seesaw problem.

Input

```
M=60 55 40
X= 5 4 5
+/(XxM)÷+/M
.774194
```

Enter weight values.

Enter distance values.

Compute center of mass.

The center of mass is .7741 feet from the center of the seesaw.

-

- performs subtraction and negation.

Case	Format	Use
	- <i>value</i>	Negation
	<i>value</i> - <i>value</i>	Subtraction
	-/ <i>array</i>	

Where *value* can be a scalar or array.

Input

Result

```
-5          ^5
-(0 ^3 7)  0 3 ^7
(13 11 9)-10 3 1 ^1
-/(100 22 25) 53
```

Notes:

1. The reduction -/*array*, is evaluated as:

$$array_1 - array_2 - \dots - array_n$$

If *array* is an empty array, the result is a 0.

2. The ^ sign (produced by $\langle \phi \rangle$ key) is used only to represent a negative number. It can not be substituted for the - function.

Application: Checkbook Balance

The example below performs a rapid checkbook balance.

Input/Result

IN=402. 15.72 62 114.49 158.25
–/IN
51.54

Enter old balance followed by each
check amount.
Compute.
New balance of \$51.54.



x multiples two (2) or more terms.

Case	Format	Use
	<i>value</i> x <i>value</i>	Multiplication
	<i>x/array</i>	Repeated multiplication

Where *value* can be a scalar or array.

Input

7 x 3	21
3x(5 8 26)	15 24 78
(13 41)x(6 8)	91 328
x/(1 2 3 4 5)	120

Note:

The reduction, *x/array* multiplies all elements of an array. It is evaluated as follows:

$$array_1 \times array_2 \times \dots \times array_n$$

If *array* is an empty-array, the result is a 1.

Application: Geometric Mean

Jenner Brooks needs to know the average monthly return for Shamrocks. His stock broker has been sending the percentage price changes for the stock over the preceding twelve months. They are:

Shamrocks
(price per share)

Percentage Change	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
	5%	2%	6%	7%	1%	0%	4%	-1%	6%	2%	5%	9%

The average monthly rate of return is the geometric mean of the growth rates, and is computed as follows:

$$\sqrt[12]{(1+.05) \times (1+.02) \times \dots \times (1+.09)}$$

The input below generates the average monthly return for Shamrocks.

Input/Result

```
PCT=5 2 6 7 1 0 4 ^1 6 2 5 9
x/(1+PCT÷100)*÷12-1
0.03792477
```

Enter percentage changes.
Compute average.
The average return is 3.8% per month.



÷ performs division and computes the reciprocal.

Case	Format	Use
	÷ <i>value</i>	Reciprocal ($\frac{1}{value}$)
	<i>value1</i> ÷ <i>value2</i>	Division ($\frac{value1}{value2}$)
	÷/array	Reduction

Where *value*, *value1* and *value2* can be scalars or arrays.

Examples:

Input	Result
÷8	.125
÷(1 2 3 4)	1 .5 .3333333 .25
12÷4	3
(10 100 25 40)÷5	2 20 5 8
(63 28)÷(9 4)	7 7
÷/(80 2 4)	10

Notes:

- 1. $0 \div 0$ gives 1, otherwise division by 0 generates an error.
- 2. The reduction, $\div/\text{array}$ is evaluated as:

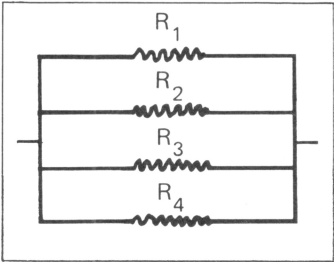
$$\text{array}_1 \div \text{array}_2 \div \dots \div \text{array}_n$$

If *array* is an empty-array, the result is a 1.

Application: Parallel Resistance

The net resistance, *R*, of *N* resistances wired in parallel is:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \dots + \frac{1}{R_N}$$



The example below generates the net resistance from the individual parallel resistances in a circuit.

Input/Result

R=15 65 33 100
÷+÷R
8.17298

Enter resistance values.
Compute net resistance.
The net resistance is 8.17 ohms.

*

* performs exponentiation.

Case	Format	Use
	<i>value1*value2</i>	Exponentiation
	<i>*/array</i>	Repeated exponentiation
Where <i>value1</i> and <i>value2</i> can be scalars or arrays		

Examples:

Input

10*2	100
(2 3 4)*3	8 27 64
5*(~1 0 2)	.2 1 25
(49 64)*(÷2,÷3)	7 4
*/10~2 3	1E~6

Notes:

1. If *value1* is zero, positive values for *value2* return 0, a zero value for *value2* returns a 1 and negative values for *value2* returns an error.
2. If *value1* is negative, non-integer values for *value2* return an error.
3. The reduction, **/array* is evaluated as follows:

$$array_1 * array_2 * \dots * array_n$$

If *array* is the empty-array, the result is 1.

Application: Evaluating a Polynomial

After extensive testing, Markus Aircraft concluded that for its newest engine, engine noise is related to rpm by the polynomial formula below:

$$db = .018x^4 + .13x^3 - .37x^2 + 4.18x + 1$$

Where db is the noise in decibels and x is thousands of revolutions per minute (rpm).

The input below computes the noise in decibels for any given engine rpm.

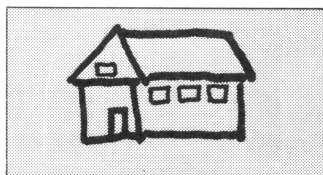
Input/Result

```
DB=.018 .13 -.37 4.18 1
X=6000÷1000
EDB=SIZE DB-INDEX SIZE DB
+/(X*EDBxDB)
64.168
```

Enter polynomial coefficients.
Enter rpm divided by 1000.
Generate exponent array.
Compute and display the noise in decibels.

Application: Mortgage Payments

Steve Lederman wants to buy an \$80,000 house in Santa Clara, CA. He can get an 80% mortgage at 10% over 30 years. What would be his monthly payments?



The house-price mortgage-payment relationship is as follows:

$$PMT = \frac{P \cdot PC \cdot R}{1 - (1+R)^{-N}}$$

Chapter 8: Alphabetical Reference Section

Where P is the house price, PC is the mortgage percentage, PMT is the payment, N is the number of payments and R is the periodic interest rate. The example below computes Steve’s monthly payment.

Input/Result

N=30x12	Calculate number of payments.
R=.10÷12	Calculate periodic interest rate.
80000x.8xR÷(1-(1+R)*-N)	Calculate payment.
561.647	Steve would pay \$561.65 per month.



The comma (,) is the concatenation function, also known as ravel. It joins scalars or arrays to return an array as a result. The comma is the upshifted character on the (N) key.

Case	Format	Use
	,value	Array creation (ravel)
	value,value	Concatenation

Where *value* can be a scalar or array.

Examples:

Input	Result
,7	An array whose element is 7.
5,16,1782	5 16 1782
15,(2x3)	15 6
(0 ^ 2 ^ 8),(15 56)	0 ^ 2 ^ 8 15 56

Note:
The case, *scalar* returns as result an one-element array containing the value *scalar*.

Application: Net Present Value

Bob Samuels invests his excess income in rental property. He uses the net present value of discounted cash flows to make his decisions.

A particular rental requires \$20,000 equity and a negative \$2400 per year to support the investment; however, Bob figures that the equity value of his investment would be worth \$60,000 in 5 years. Bob wants to earn 15% of his cash per year. Is this investment worthwhile?

The program below calculates the net present value of a stream of cash flow.

```

PROG NPV
"INITIAL EQUITY"
BEQ=KEYB
"ON-GOING CASHFLOW"
CF=KEYB
"ENDING EQUITY"
EEQ=KEYB
R=1.15
CF=BEQ,(ARRAY 5+CF),EEQ
I=INDEX 7-1
"NPV AT 15%"
+/(CF÷(R*I))
ENDP

```

Enter beginning equity.

Enter ongoing cashflow.

Enter ending equity.

Discount rate of 15%.

Create cashflow array.

Create index array.

Display net present value.

The session below analyzes Bob's investment opportunity.

Input/Result

```

NPV
INITIAL EQUITY
? 20000
ON-GOING CASHFLOW
? 2400
ENDING EQUITY
? 60000
NPV AT 15%
2105.469

```

/

The slash (/) is the array reduction function. In conjunction with two argument functions, it operates on all the elements of an array to reduce the array to a scalar. The slash is the upshifted character on the <H> key.

Case	Format	Use
	function/value	Array reduction

Where function is any two argument function (except the concatenation function (,), and value is any scalar or array.

Used with arrays, array reduction evaluates the expression that results from placing *function* between the successive elements of the array as follows:

$value_1 \text{ function } value_2 \text{ function } \dots \text{ function } value_N$

For example, the addition function (+) can be used in an array reduction as follows:

Input	Result
+/1 2 3 4	10

The procedure can be visualized as expanding the above input into the expression, “1+2+3+4”, which evaluates to the value of 10. Further examples of array reduction follows:

Input	Result
J=15 3 4	no display
-/J	8 Repeated subtraction.
x/(1 2 3 4)	24 Product.
MAX/6 7 10 11	11 Maximum.
*/ 2 2 2	16 Repeated exponentiation.
COMP=7 7	no display
GE/COMP	1 Greater than or equal to'

When *value* is a scalar, the result of applying array reduction is the scalar value.



! computes the factorial.

Case	Format	Use
	!value	Factorial
Where <i>value</i> represents a non-negative integer or an array of non-negative integers.		

Examples:

Input	Result
!5	120
!7 0 3	5040 1 6

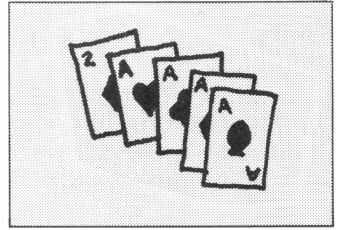
Notes:

- 1. Negative values and non-integer values of *value* generate errors.
- 2. The result of !0 is 1.
- 3. The factorial is computed according to the following formula:
1x2x... xN

Application: Combinatorics

Roberto, a weekend poker player, wants to know the possible five card poker hands. To generate this value, he utilizes the formula for the number of combinations of n objects taken m at a time; where n is 52 the number of playing cards, and m is 5 the size of a draw poker hand.

$$\binom{n}{m} = \frac{n!}{(n-m)!m!} = \frac{nx(n-1)x \dots (n-m+1)}{m!}$$



The input below performs Roberto's computation.

Input/Result

```
N=52
M=5
x/(INDEX M+N-M)÷!M
2598960
```

Enter total number of cards.
Enter number of cards used at a time.
Compute and display number of unique combinations.

ABS

ABS generates the absolute value.

Case	Format	Use
	ABS <i>value</i>	Absolute value

Where *value* can be a scalar or array.

Examples:

Input	Result
ABS 20	20
ABS(7 0 7 5)	7 0 7 5

Application: Mean Absolute Deviation

The formula to compute the mean absolute deviation, MAD, is as follows:

$$MAD = \sum \frac{|x - \bar{x}|}{N}$$

Where x is one of N statistical values, $\bar{x}$ is the mean of all values and N is the number of values.

The input below translates this formula in APL/S statements.

Input/Result

```
ROW=10 19 25 12 13
MEAN=+/ROW÷SIZE ROW
+/ABS(ROW-MEAN)÷SIZE ROW
4.96
```

Enter values.
Compute mean (MEAN=15.8)
Compute and output MAD.

AND

AND is a logic function. If both of two conditions are true (have the value 1), the result is true (value of 1). Otherwise the result is false (value of 0).

Case	Format	Use
	<i>value AND value</i>	Both true.
	AND/ <i>array</i>	All true.

Where *value* represent the scalars, 1 or 0, or are arrays whose elements are 1 or 0.

Examples:

Input	Result
<pre>0 AND 1 (0 1 1)AND(0 1 0) AND/1 1 1 (6 GT 2)AND(4 EQ 10)</pre>	<pre>0 0 1 0 1 0</pre>

Notes:

1. If *value* is not a 0 or 1, an error is displayed.
2. The reduction, AND/*array* compares all elements of the *array* as follows:
 $array_1 \text{ AND } array_2 \text{ AND } \dots \text{ AND } array_n$
If *array* is the empty-array, the result is a 1.

Application: Compound Conditions

The AND function can be used to form compound conditions in IF and WHILE statements. A sample WHILE statement follows.

```
WHILE (I LT 50) AND (J LT I)      WHILE with compound condition.
DO ...
```

Application: All Positive Elements

The example below generates a 1 if all the elements of the array ROW are greater than 0 and generates a 0 otherwise.

Input

```
ROW=7 7 7 5 6
AND/(ROW GT 0)
```

Enter array.
Determine and display result.

ARRAY

ARRAY generates an array whose elements are initialized to 0.

Case	Format	Use
	ARRAY <i>value</i>	Array creation.

Where *value* is a non-negative integer and indicates the size of the array.

Input

Result

ARRA 3	0 0 0
ARRA 1	An array whose element is 0.
ARRA 0	The empty array.
ARRA 5+7	7 7 7 7 7

Notes:

1. If *value* is not a positive integer scalar or one element array, an error results.
2. If the *value* is 0, the empty-array is created. The empty-array is an array with no elements. The empty-array has a size of zero, and is generated by the following situations:

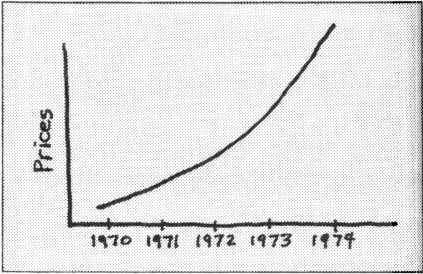
```
SIZE      scalar
INDEX     0
ARRAY     0
```

The empty array is displayed as a blank line. Use of the empty array as inputs to functions is summarized in the introduction to this section on functions.

Application: Budgeting

The Melmons want to develop a family budget. They describe their expected expenses as follows:

- Food = \$450 per month and escalating at \$10 per month.
- Clothing = \$60 per month; \$200 extra in August for school clothes; \$200 extra in November for winter clothing.
- Housing = \$1,000 per month.
- Auto = \$150 per month.
- Major = \$1000 in March and \$300 in May.
- Other = \$300 per month.



Mr. and Mrs. Melmon make \$2,100 per month after taxes. In June, their income will increase to \$2,200 per month. Will they save money this coming year and which months will cause cash problems?

The example below allows the Melmons to develop a budget.

Input/Result

```
FOOD=INDEX 12x10+450
CLO=ARRAY 12+60
CLO(8)+CLO(8)+200
CLO(11)=260
HOUS=ARRAY 12+1000
AUTO=ARRAY 12+150
MAJOR=ARRAY 12
MAJOR(3)=1000
MAJOR(5)=300
MISC=300+ARRA 12
IN=( INDEX 12 GT 6)x100+2100
SAV=IN-(FOOD+CLO+HOUS+
AUTO+MAJOR+MISC)
SAV
130 120 ^890 100 ^210 80
170 ^40 150 140 ^70 120
+/SAV
^200
```

- Define food budget.
- Define clothing budget.
- Change August budget.
- Change November budget.
- Define housing budget.
- Define auto budget.
- Define major expenses budget.

- Define other expenses.
- Define income stream.

- Display monthly cashflows.

- Compute and display savings.
- The Melmons net savings is a negative \$200.00 for the year.

CEIL

CEIL gives the lowest integer greater than or equal to the specified value.

Case	Format	Use
	CEIL <i>value</i>	Integer ceiling

Where *value* represents a scalar or array.

Examples:

Input	Result
CEIL 7.3	8
CEIL (2 ⁻⁸ 8.9 3.006)	2 ⁻⁸ 4

Notes:

1. If *value* is an integer, the integer is returned; otherwise, the next larger integer is returned.
2. If *value* is within 3.76×10^{-9} of an integer value, the integer value is returned.

Application: Rounding

The statements below compute the result of any value, rounded to the nearest N decimal places.

Input/Result

```
N=10 1 .1 0.1
VAL=37.3943
CEIL(VAL÷N-.5)×N
40 37 37.4 37.39
```

Enter N.
Enter value to be rounded.
Compute rounded result.
The value 37.3943 rounded respectively
to the nearest 10, 1, .1, and .01.

COS

COS computes the cosine.

Case	Format	Use
	COS <i>value</i>	Cosine

Where *value* can be a scalar or array.

Examples:

Input	Result
$\text{COS } 0$	1
$\text{COS}(\pi, (\pi \div 2), (\pi \div 4))$	$-1 \ 0 \ .707107$
$\text{COS}(25 \times \pi \div 4, 1)$	$.707107 \ .540302$

Notes:

1. *Value* must be expressed in terms of radians (90° is equal to $\frac{\pi}{2}$ radians).
2. The value of π in APL/S is 3.141593.

Application: Polar Coordinates

The example below converts radius and degrees to Cartesian coordinates using the following formulas.

$$x = r \cdot \cos \theta$$
$$y = r \cdot \sin \theta$$

Where r is the radius, θ is the angle in degrees, x is the horizontal cartesian coordinate and y is the vertical cartesian coordinate.

Input/Result	
R=10	Enter radius.
D=50	Enter degrees.
T=D÷180×π	Compute radian equivalent.
R×(COS T,SIN T)	Generate and output x and y.
6.42788 7.66044	

EQ

EQ compares two values for equivalence. The result is 1 if both values are equal and 0 otherwise.

Case	Format	Use
	<i>value EQ value</i>	Equivalence
	<i>EQ/value</i>	Reduction

Where *value* represents scalars or arrays.

Examples:

Input	Result
7 EQ 11	0
(18 1004) EQ (18 1103.9)	1 0
EQ/(2x3,18÷3)	1

Notes:

1. The case EQ/array generates a meaningful result only where array contains 2 elements. If array is the empty-array, the result is 1.
2. EQ performs a fuzzy compare. If the absolute percentage difference between the input values is less than 3.76E-9, then the result is a 1.

Application: Julian Calendar Program

The Julian Calendar Program expresses dates as an integer, calculated as the number of days since January 1, 1900. The program is useful for computing the number of days between two specified dates. The program below computes the Julian date for March 31, 1984.

```

PROG CTOJ
Y=1984
M=3
D=31
OUT=(Y-1900)x365+FLOO(Y÷4)+
Mx31-+/(M EQ 2 2 2 4 6 9 11)+D
ENDP

```

EXP

EXP computes the exponential.

Case	Format	Use
	EXP <i>value</i>	Exponential (e^x)

Where *value* represents a scalar or array.

Examples:

Input	Result
EXP 1	2.718282
EXP(0 2 ^2)	1 7.38906 0.1353353
EXP(LOG 17)	17

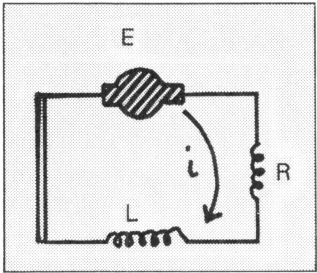
Note:

The exponential value of 1 in APL/S is 2.718282.

Application: LR Circuit

Ohm's law, $E = Ri$, requires modification in a circuit containing self-inductance. The modified form is:

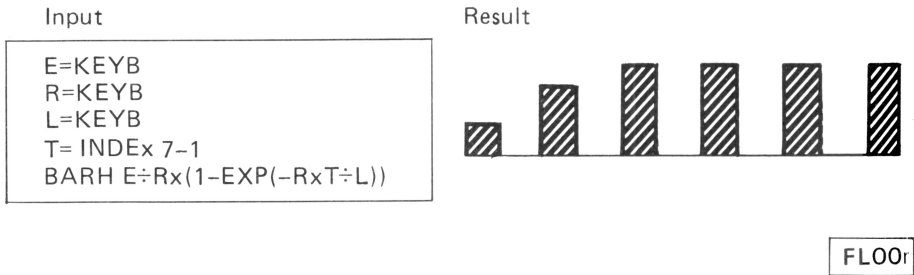
$$L \frac{di}{dt} + Ri = E$$



Where R is the resistance in ohms L is the self-inductance in henries, E is the impressed electromotive force (emf) in volts and i is the current in amperes. After integration, the relationship between current flow and time can be expressed as:

$$i = \frac{E}{R} (1 - e^{-Rt/L}) \quad t = 0, 1, 2 \dots N$$

The example below graphs the current versus time relationship.



FLOOR gives the largest integer less than or equal to the specified input.

Case	Format	Use
	FLOOR <i>value</i>	Integer floor
Where <i>value</i> represents a scalar or array.		

Input	Result
FLOOR 30.95	30
FLOOR (16 83.19 ^83.19)	16 83 ^84

Notes:

1. If *value* is an integer, the integer is returned; otherwise, the next lower integer is returned.
2. If *value* is within $3.76E^{-9}$ of an integer value, the integer value is returned.

Application: Significant Digits

Regina is in the business of prediction, but the accuracy of her predictions is limited by both equipment and methods. Regina has deduced that her predictions are always accurate to 4 significant digits, and so modifies her predicted results.

The input below generates results accurate to four digits.

Input/Result

```
VAL=25 10 2 .07
RSLT=6xEXP(.3xVAL)
N=4

S=10*(FLOOR(10 LOG RSLT)-N+1)
FLOOR(RSLT÷S+.5)xS
10850 120.5 10.93 6.127
```

Enters initial data.
 Regina's secret formula.
 Enter desired number of significant digits.
 Compute decimal shift factor.
 Shift, compute, and display result.

GE

GE compares two values for the greater than or equal to condition.

Case	Format	Use
	<i>value1</i> GE <i>value2</i>	Greater than or equal to
	GE/array	Reduction

Where *value1*, *value2* represent scalars or arrays.

Input	Result
7GE 4	1
(3 1) GE 1	1 1
(17 59)GE(14 100)	1 0
GE/(5*2,25)	1

Notes:

- 1. The reduction, *GE/array* generates a meaningful result only when *array* has 2 elements. If *array* is the empty-array, the result is 1.
- 2. GE performs a fuzzy compare. If the absolute percentage difference between the input values is less than 3.76E-9, then the result is a 1.
- 3. If *value1* is greater than or equal to *value2*, then the result is a 1, otherwise a 0.

GT

GT compares two values for the greater than relationship.

Case	Format	Use
	<i>value1</i> GT <i>value2</i>	Greater than
	GT/ <i>array</i>	Reduction

Where *value1* and *value2* represent scalars or arrays.

Examples:

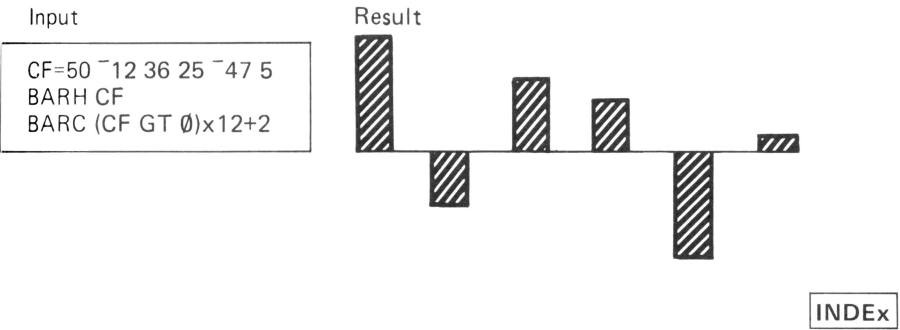
Input	Result
9 GT 10	0
(^-3 5 1 2)GT 1	0 1 0 1
(100 101 1)GT(101 101 0)	0 0 1
GT/(26,12x2)	1

Notes:

- 1. The reduction, *GT/array*, is meaningful only when *array* contains 2 elements. If *array* is the empty-array, the result is 0.
- 2. GT performs a fuzzy compare. If the absolute percentage difference between the input values is less than 3.76E-9, then the result is 0.
- 3. If *value1* is greater than *value2* then the result is a 1, otherwise a 0.

Application: Profit or Loss

Businessmen speak of profits as “in the black,” and of losses as “in the red.” The example below sets the negative bars to the red color (2) and positive bars to the yellow color (14).



INDEX creates an indexed array. The function generates an array containing successively all the integers from 1 to and including the function input.

Case	Format	Use
	INDEX <i>value</i>	Create indexed array
Where <i>array</i> must be a positive integer.		

Examples.

Input	Result
INDEX 7	1 2 3 4 5 6 7
INDEX 2+3	4 5
INDEX 1	An array containing as element, 1
INDEX 0	The empty array.

Notes:

- 1. If *value* is 0, the result is the empty array.
- 2. If *value* is not a positive integer, an error is generated.
- 3. If *value* is not a scalar or 1 element array, an error is generated.

Application: Discounted Cashflows

Dr. Haynes has his own model for evaluation of income stocks. He forecasts the expected dividend payments of a company for the next five years and computes the net present value of discounted cashflows. The model appears below:

$$V = \sum_{i=1}^4 \frac{D_i}{(1+R)} + \frac{D_5}{\frac{R-G}{(1+R)^5}}$$

Where D are the dividend forecasts, R is the discount rate and G is the long term economic growth rate.

For example, Dr. Haynes forecasts the following dividend payments for Tahitian Electric:

	1	2	3	4	5
Dividends per share	3.50	4.00	4.50	5.00	5.50

For this type of company, Dr. Haynes uses a discount rate of 15% and an expected growth rate of 10%. The program below computes the value of a stock using Dr. Haynes' Model.

```
PROG DCF
"ENTER DIVIDENDS"
D=KEYB
N=SIZE D
D(N)=D(N)÷(.15-.10)
I=INDEX N
"EXPECTED VALUE"
+/(D÷(1.15*I))
ENDP
```

Accept dividend values.
Determine N.
Compute terminal value.
Generate index array.

Discount, sum and display.

Input/Result

```
DCF
ENTER DIVIDENDS
? 3.5 4 4.5 5 5.5
EXPECTED VALUE
66.5752
```

Run program.

Enter dividend values.

Answer

LE

LE compares two values for the less than or equal to condition.

Case	Format	Use
	<i>value1</i> LE <i>value2</i>	Less than or equal to
	LE/ <i>array</i>	Reduction

Where *value1* and *value2* represent scalars or arrays.

Examples:

Input	Result
6 LE 8	1
(25 12 96) LE 25	1 1 0
(6 8 9) LE (6 7 8)	1 0 0
LE/(4 5)	1

Notes:

1. The reduction, LE/*array*, is meaningful only when array contains 2 elements. If *array* is the empty-array, the result is 1.
2. LE performs a fuzzy compare. If the absolute percentage difference between the input values is less than 3.76E-9, then the result is 1.
3. If *value1* is less than or equal to *value2*, then the result is a 1, otherwise a 0.

LT

LT compares two values for the less than condition.

Case	Format	Use
	<i>value1</i> LT <i>value2</i>	Less than
	LT/ <i>array</i>	Reduction

Where *value1* and *value2* represent scalars or arrays.

Examples:

Input	Result
43.00001 LT 43.00002	1
(29 17 3)LT 6	0 0 1
(15 5)LT(20 4)	1 0
LT/(-6 -5)	1

Notes:

1. The reduction, *LT/array*, is meaningful only when array contains two elements. If *array* is the empty array, the result is 0.
2. LT performs a fuzzy compare. If the absolute percentage difference between the input values is less than 3.76E-9, then the result is 0.
3. If *value1* is less than *value2*, then the result is 1, otherwise a 0.

LOG

LOG computes the natural and base N logarithms.

Case	Format	Use
	LOG <i>value</i>	Natural logarithm
	<i>value1</i> LOG <i>value2</i>	Base logarithm
	LOG/ <i>array</i>	Reduction

Where *value*, *value1*, *value2* can be scalars or arrays of positive values.

Examples:

Input	Result
LOG 10	2.30259
10 LOG 100	2
(10 5 2) LOG 1000	3 4.29202 9.96579
(9 3) LOG (729 81)	3 4
LOG/(10 100 128)	7

Notes:

1. *Value1* is the base of the logarithm.
2. If *value1* is 1, then *value2* must equal 1 and the result is a 1, otherwise the result is undefined.
3. The reduction, LOG/*array* is evaluated as follows:

$$\text{array}_1 \text{ LOG } \text{array}_2 \text{ LOG } \dots \text{ LOG } \text{array}_n$$
 If *array* is an empty-array, the result is an error.

Application: Payment on a loan

Michael Peak wants to purchase a new car costing \$4988. He can afford just \$150 down and \$150 per month. If the yearly interest rate is 12%, how many payments would he have to make? To compute the number of payments see the formula below:

$$PV = PMT \frac{1 - (1 + I)^{-N}}{I}$$

Where N is the number of payments, PMT is the periodic payment amount, PV is the present value of the loan, I is the periodic interest rate, the equation can be solved for N as follows:

$$\frac{PMT}{(PMT - PV \times I)} = (1 + I)^N$$

Input/Result

PV=4988-150
 PMT=150
 I=.12÷12
 (1+I)LOG(PMT÷(PMT-PV×I))
 39.13388

Enter amount yet to be paid.
 Enter monthly payment.
 Enter interest rate per month.
 Compute and display number of monthly payments.
 39 months of payments.

MAX

MAX returns the maximum of two or more values.

Case



Format

value MAX *value*

Use

Larger of two values



MAX/*array*

Largest value in array

Where *value* can be a scalar or array.

Examples:

Input	Result
6 MAX 7	7
(7 10 13) MAX 10	10 10 13
(8 4) MAX (2 7)	8 7
MAX/(1 2 3 0)	3

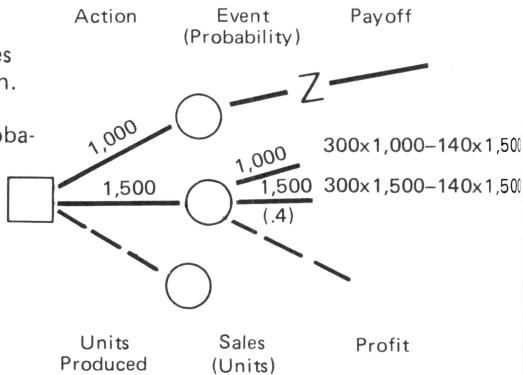
Note:

1. The reduction, MAX/array retains the largest value in array. If array is the empty-array, the result is $\overline{3.618502E75}$, the smallest number in APL/S.

Application: Decision Analysis

The LSI Computer Company produces novelty items for the Christmas season. A particular product is sold for \$300. Management assigns the following probabilities to various levels of sales:

Sales	Probability
1,000	.1
1,500	.4
2,000	.3
2,500	.1
3,000	.1



The cost of manufacturing this item varies with the number produced, as shown below:

Units Produced	Average Cost per Unit
1,000	\$180
1,500	140
2,000	115
2,500	110
3,000	90

If items are not sold during Christmas, the value of the excess product is zero. How many units should be produced? What is the expected profit?

Decision sciences shows that LSI should choose to produce the number of units that have the highest expected monetary value. The program below solves the problem.

```

PROG DEC
P=.1 .4 .3 .1 .1
U=S=1000 1500 2000 2500 3000
C=180 140 115 100 90
I=1
EMV=ARRA 5
WHIL I LE 5
DO SELL=S MIN U(I)
    PAYO=300xSELL-C(I)xU(I)
    EMV(I)=+/(PAYOxP)
    I=I+1
ENDW
(EMV EQ MAX/EMV)xU
MAX/EMV
ENDP

```

Enter probabilities.
Units produced and sales.
Average cost per unit.

Expected monetary values.
For each action by mgt.
Compute units sold.
Compute payoff.
Compute EMV.

Display units produced.
Display expected profit.

MIN

MIN returns the minimum of two or more values

Case	Format	Use
	<i>value</i> MIN <i>value</i>	Smaller of two values
	MIN/ <i>array</i>	Smallest value in array

Where *value* represents a scalar or array.

Examples:

Input	Result
25 MIN 19	19
(2 8 40) MIN 8	2 8 8
(7 11)MIN(6 5)	6 5
MIN/(5 25 2 6)	2

Note:

1. The reduction, MIN/*array* returns the smallest value in *array*. If *array* is the empty-array, the result is 3.618502E75, the largest number in APL/S.

Application: Prime Numbers

The program below generates the first N prime numbers, where N is a program input.

```

PROG GEN
PRIME=,2
NEW=3
N=KEYB

WHILE SIZE PRIME LT N

DO TEST=MIN/(NEW MOD PRIME)

    IF TEST NE 0
    THEN PRIME=PRIME,NEW
    ENDI
ENDW
PRIME
ENDP

```

First prime.
First number to test for prime.
Accept number of primes from the keyboard.
Continue until N primes are generated.
The smallest remainder from the division by previous primes.
If remainder not zero.
Then concatenate new prime to PRIME list.

Display prime numbers.

Input/Result

```

GEN
?10
2 3 5 7 11 13 17 19 23 29

```

Run program.
Enter desired number of primes.
Display the prime numbers.

MOD

MOD computes the remainder from the division of two values.

Case	Format	Use
	<i>value1</i> MOD <i>value2</i>	Remainder from $value1 \div value2$
	MOD/ <i>array</i>	Reduction

Where *value1* and *value2* represent scalars or arrays.

Examples:

Input	Result
72 MOD 10	2
(16 10 6 ^1)MOD 3	1 1 0 2
8 MOD(3 ^3 2)	2 1 0
(1.5 .873)MOD 1	.5 .873
(5 8)MOD(8 5)	5 3
MOD/(100 7)	2

Notes:

- 1. The result of *value1* MOD *value2* is computed from the following expression:
 $result = value1 - value2 \times FLOOR(value1 \div value2)$
- 2. If *value1* is 0, the result is *value1*.
- 3. The reduction, MOD/*array* is evaluated as follows:
 $array_1 \text{ MOD } array_2 \text{ MOD } \dots \text{ MOD } array_n$
If *array* is the empty-array, the result is $\emptyset$.

Application: Array Shift

The program below shifts values in a 7 element array J-positions to the left.

```
PROG SHIFT
I=0
OUT=IN
WHILE (I=I+1) LE 7
DO OUT(I)=IN((I+J-1)MOD 7+1)
ENDW
ENDP
```

Input/Result

```
IN=INDEX 7
BARH IN
J=2
SHIFT
BARH OUT
```

NE

NE compares two values for the not equal to condition.

Case	Format	Use
	<i>value1</i> NE <i>value2</i>	Not equal to
	NE/ <i>array</i>	Reduction

Where *value1* and *value2* represent scalars or arrays.

Examples:

Input	Result
10 NE 7	1
(5 4 8 20) NE 5	0 1 1 1
(3.5 7.8) NE (3.5 7.5)	0 1
NE/(1.0006 1.00062)	1

Notes:

1. The reduction `NE/array`, is meaningful only when `array` contains two elements. If `array` is the empty-array, the result is 0.
2. NE performs a fuzzy compare. If the absolute percentage difference between the input values is less than $3.76E^{-9}$, then the result is 0.
3. If `value1` is not equal to `value2` then the result is a 1, otherwise a 0.

NOT

NOT returns the complementary result of a logical expression.

Case	Format	Use
→	NOT <i>value</i>	Not true
Where <i>value</i> represents either 0 or 1, or is an array whose elements represent 0 or 1.		

Examples:

Input	Result
NOT 1	0
NOT(6 GT 7,(13 LE 5))	1 0

Notes:

1. If `value` is not a 0 or 1, an error is displayed.
2. If `value` is a 0, the result is a 1; if `value` is a 1, the result is a 0.

OR

OR is a logic function. If either of two conditions is true (has the value 1), the result is true (value of 1). Otherwise the result is false (value of 0).

Case	Format	Use
	<i>value OR value</i>	Either true
	OR/array	Any true
Where <i>value</i> represents either the scalars 1 or 0, or arrays whose elements are 1 or 0.		

Examples:

Input	Result
1 OR 0	1
(6 GT 10) OR (5 NE 5)	0
(1 0 0) OR 1	1 1 1
(1 0 1) OR (0 1 1)	1 1 1
OR/(0 0 0 0 0 0)	0

Notes:

- 1. If *value* is not a 0 or 1, an error is displayed.
- 2. The reduction, OR/array compares all elements of the *array* as follows:
array(1) OR array(2) OR . . . OR array(n).
If *array* is the empty-array, the result is a 0.

Application: List Search

In programming, it is often necessary to compare a value against a list of known values. The sequence below demonstrates a simple search through a list of values.

The problem is to evaluate the formula $\frac{1}{x(x^2-1)}$ for any value of *x*. If *x* is equal to -1, 0, or 1, the formula evaluates to infinity.

<pre>PROG SAMP X=KEYB IF OR/(X EQ -1 0 1) THEN "INFINITY" ELSE ÷(X*3-X) ENDI ENDP</pre>	Accept value for <i>x</i> . List search.
---	---

RAND

RAND generates a random integer. The integer generated can range between 1 and the value of the function input inclusively.

Case	Format	Use
	RAND <i>value</i>	Random integer
Where <i>value</i> represents a positive integer or an array of positive integers.		

Examples:

Input	Result
RAND 7	6
RAND 7	1
RAND 7	3
RAND(100 3)	95 2
RAND(100 3)	18 2
RAND(100 3)	42 3

Notes:

- 1. If *value* is not a positive integer, an error is displayed.
- 2. RAND draws a random number from the uniform distribution.
- 3. The initial seed value after MASTER CONTROL is random.
- 4. The current seed value is saved on tape by a SAVE.

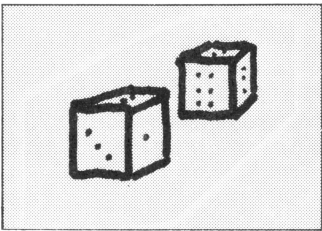
Application: Random Values

The example below generates N 4 digit random values with the range of 0 to 1. These numbers are useful for random sampling and statistical testing.

Input	
N=30 R=RAND(ARRAY N+1000)÷1000	Define N. Create variable R containing 30 random values.

Application: Throwing Dice

The following program simulates the throw of two dice and displays the throw as bars. A three and a six would appear as three tall bars and three short bars. Double four would be displayed as four tall bars.



```

PROG DICE
R=RAND 6 6
B=INDEX 6
BARH(R(1) GE B+R(2) GE B)x32
ENDP

```

Generate two random values.
Create index array.
Create and scale bars.

SIGN

SIGN generates the sign value of the input value.

Case	Format	Use
	SIGN <i>value</i>	Signum

Where *value* represents a scalar or array.

Examples:

Input	Result
SIGN 125	1
SIGN(-50 -.005 0 .0002 18)	-1 -1 0 1 1

SIN

SIN computes the sine.

Case	Format	Use
	SIN <i>value</i>	Sine

Where *value* represents a scalar or array.

Examples:

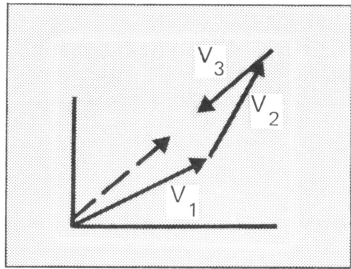
Input	Result
SIN($\pi\div 2$)	1
SIN($3\times\pi\div 2$),($\pi\div 4$)	-1 .707105
SIN(0 1 2)	0 .841471 .909297

Notes:

- 1. *Value* must be expressed in terms of radians (90° is equal to $\frac{\pi}{2}$ radians).
- 2. The value of π in APL/S is 3.141593.

Application: Vector Summation

The program computes the sum of N vectors. The inputs are expressed in polar terms (radius, angle) and the result is given in rectangular terms (x,y).



```
PROG VECA
R=10 5 6
A=( $\pi\div 6$ ),( $\pi\div 3$ ),( $3\div 2\times\pi$ )
+/(RxCOS A),+/(RxSIN A)
```

Enter radius values.
Enter angle values.
Compute and display rectangular coordinates of resulting vector.

SIZE

SIZE counts the number of elements in the input array.

Case	Format	Use
	SIZE <i>array</i>	Array size

Examples:

Input	Result
SIZE(5 11 3 8 2)	5
ROW=7	
SIZE ROW	1
SIZE 7	The empty array.
SIZE ARRA $\emptyset$	$\emptyset$
SIZE(ROW,(10 13))	3

Notes:

1. If the input to size is a scalar, the result is the empty-array.
2. If *array* is the empty-array, the result is $\emptyset$.
3. The result is itself a 1 element array.

TAN

TAN computes the tangent.

Case	Format	Use
	TAN <i>value</i>	Tangent
Where <i>value</i> represents a scalar or array.		

Examples:

Input	Result
TAN($\pi \div 4$)	1
TAN(0, ($\pi \div 3$), π)	0 1.73205 0

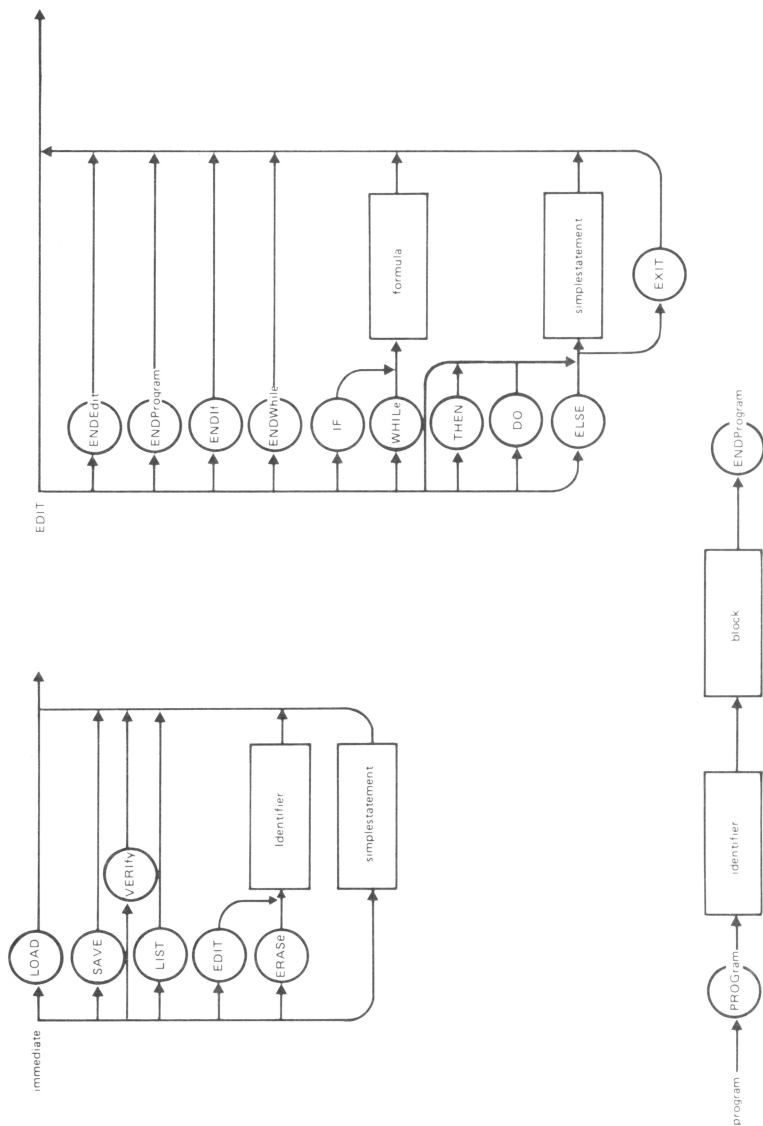
Notes:

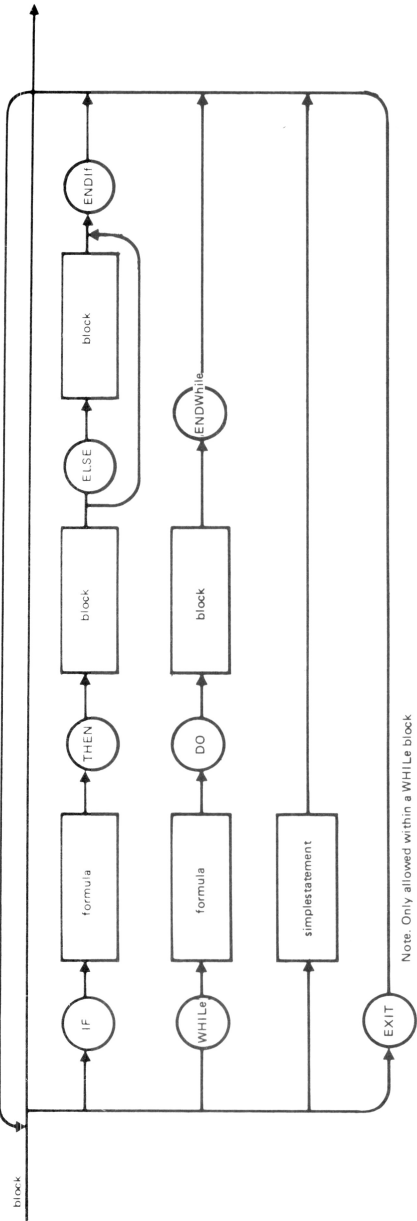
1. *Value* must be expressed in terms of radians. (90° is equal to $\frac{\pi}{2}$).
2. The value of π in APL/S is 3.141593.

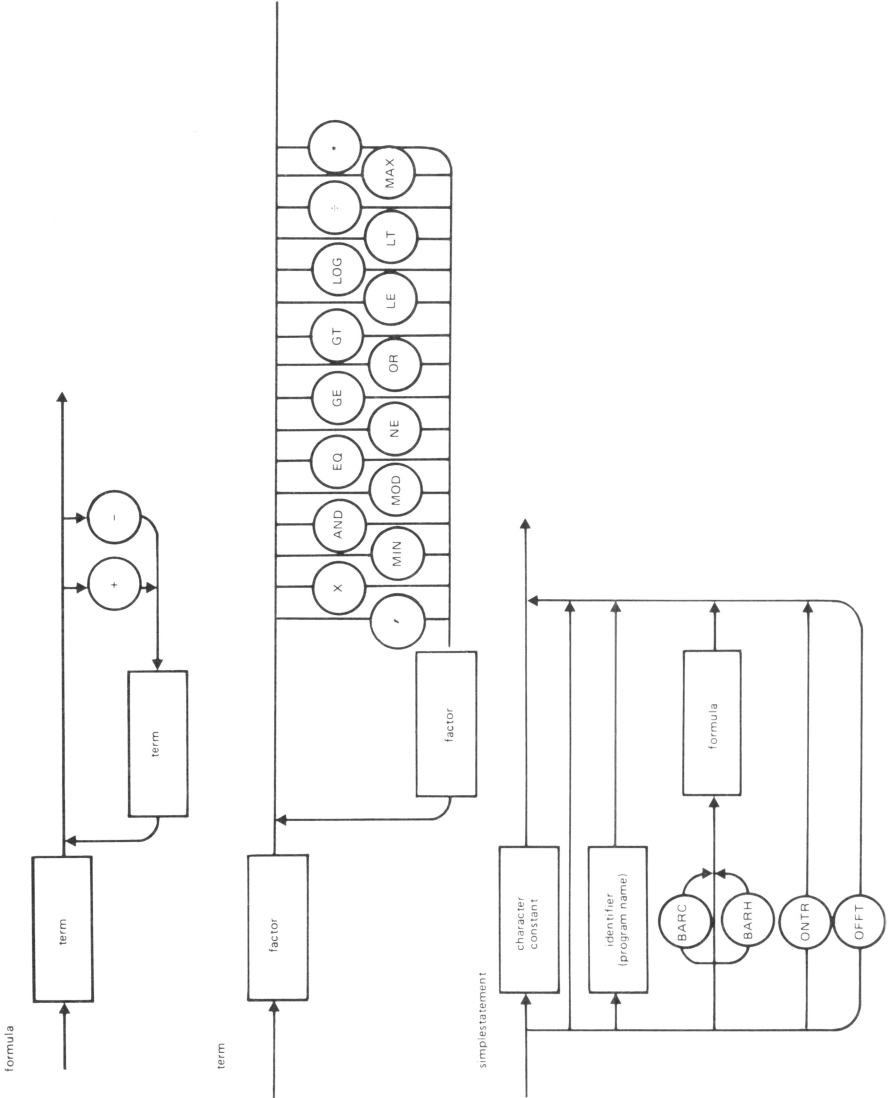
Appendices

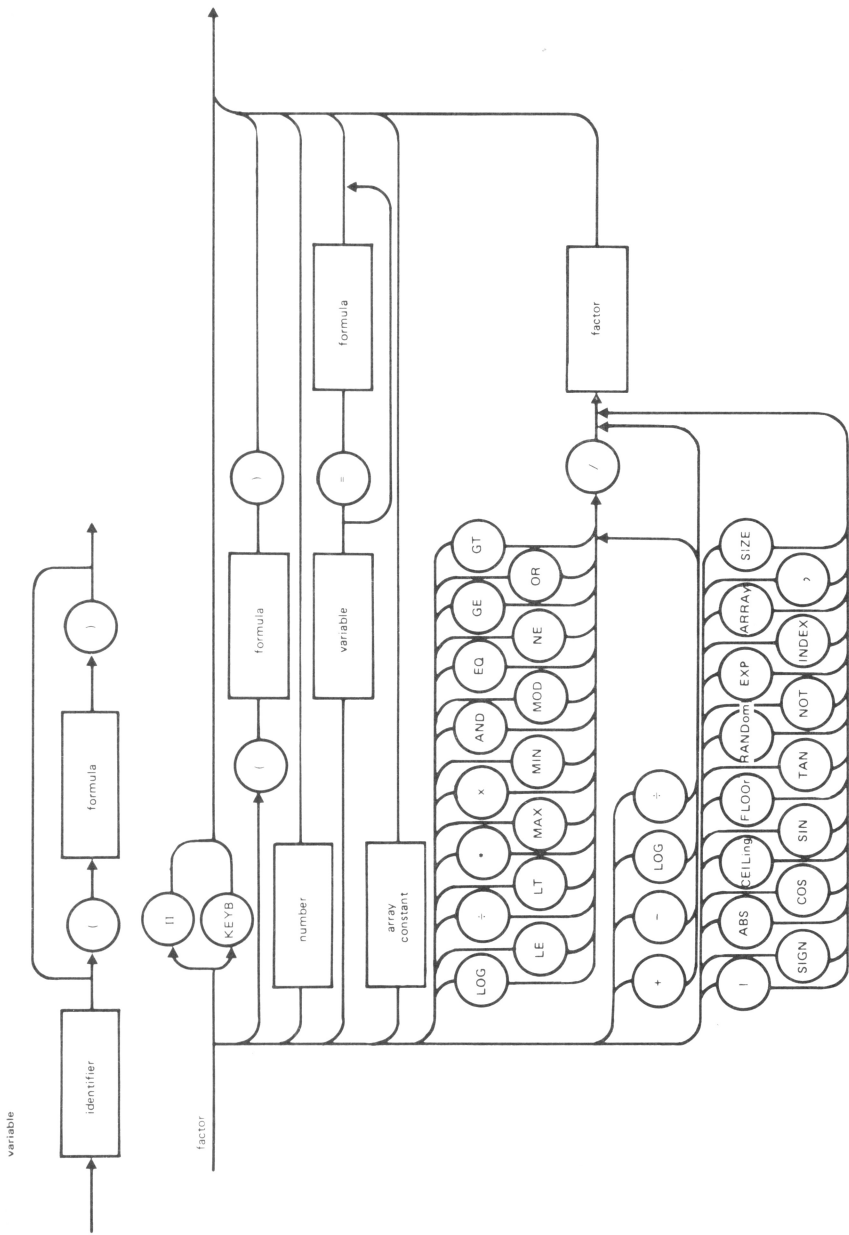
Appendix A: Syntax Diagrams

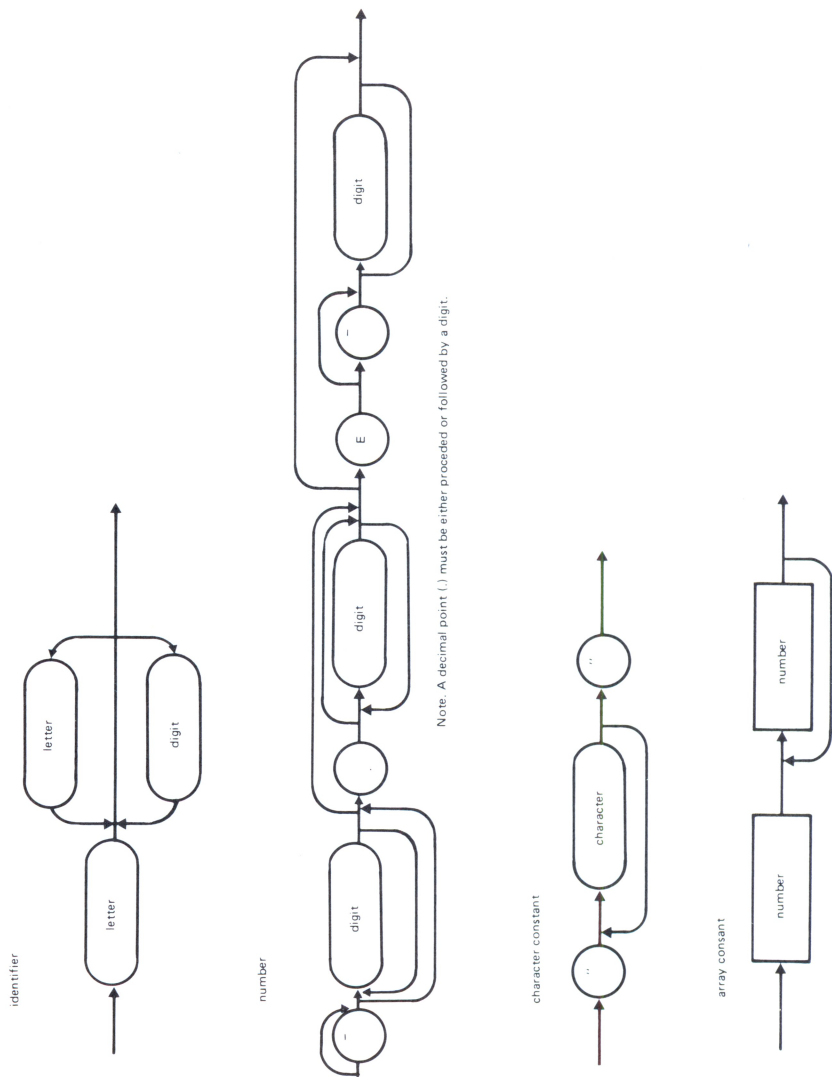
The following pages express the APL/S language using a syntax diagram. Starting at the diagrams labelled Immediate, Edit, and Program, a path through the diagram defines a syntactically correct immediate mode statement, edit mode statement and program respectively. The diagrams do not specify APL/S action after the statement is entered. Each box references a diagram by that name; and the diagram defines its meaning. Each rounded enclosure specifies terminal symbols, which are key words and symbols that are actually typed into the computer.











Appendix B: Technical Notes

Values:

All values are stored in 4-byte floating point.

Range of numbers that can be input is $\pm 3.618502 \pm 75$.

Range of number that can be represented internally and output is $\pm 3.618502E^{-79}$ to $\pm 3.618502E^{+75}$.

Precision is 6 to 7 digits.

Display:

64 character text display in 4 physical lines.

125 character maximum logical line.

7 bars for graphics (16 colors).

Memory:

1K bytes of RAM for TV support and internal buffers.

1K bytes of RAM for program and array storage.

— keywords are stored as single byte tokens.

— variable references are stored as single byte tokens.

Appendix C: Error Messages

When APL/S encounters an error, the error message includes the type of error and an error number. This appendix lists APL/S errors by type and by number.

Value (Evaluation error)

1. An attempt has been made to use the value of a variable for which no value has been assigned.
2. An evaluated number is too small or too large. (Floating point overflow in a scalar function).
3. A two-argument function has been applied to two arrays of unequal length.
4. The reduction of an empty-array has no identity value.
5. Subscripted variable is not an array.
6. Subscript is out of range or non-integer.
7. Argument to INDEX or ARRAY is not between the values of 0 and 255 or not an integer.
8. Scalar or one element array is required.
9. Value of $\emptyset$ or 1 is required.
10. Attempt has been made to edit a variable. Only programs can be edited.
11. Attempt has been made to execute a program that is not executable. (Program has a syntax error).
12. For $value1 \text{ MOD } value2$, $value1$ exceeds $value2$ by too great an amount. For SIN, COS, or TAN, the argument is too large.
13. Factorial requires a non-negative integer argument less than 128.
14. The value $\emptyset$ cannot be raised to a negative power.
15. For LOG, one of its arguments is less than or equal to zero.
16. A positive non-zero integer argument is required for RAND.
17. For $value1 * value2$, $value1$ is less than zero and $value2$ is not an integer.

Syntax (Errors in statement construction)

3. An EXIT was found outside a WHILE block.
5. A program-name or variable-name must follow the ERASE or EDIT command.
6. Following what appeared to be a valid immediate mode statement additional characters were found.
13. BARH or BARC must be followed by a valid formula.
17. Failed to find a valid term on the right of a two-argument + or – function.

23. Failed to find a valid factor on the right of a two-argument LOG, $\div$, $\ast$, $\times$, AND, EQ, GE, GT, LE, LT, MAX, MIN, MOD, NE, or OR, function.
24. Failed to find a valid factor on the right of a concatenation $\langle, \rangle$ function.
35. Failed to find the reduction operator $\langle / \rangle$ following a LOG, $\div$, $+$, $\times$, AND, EQ, GE, GT, LE, LT, MAX, MIN, MOD, NE, or OR.
38. Failed to find a valid factor following the reduction operator $\langle / \rangle$.
39. Failed to find a valid factor following a one-argument $+$, $-$, LOG, $\div$, $!$, ABS, CEIL, COS, EXP, FLOOR, NOT, RAND, SIGN, SIN, or TAN function.
40. Failed to find a valid factor following a one-argument ARRAY, INDEX, SIZE, or ravel $\langle, \rangle$ function.
41. Failed to find a valid formula following a left parenthesis.
42. Failed to find a right parenthesis, not used for subscripting.
45. Failed to find a valid formula following a left parenthesis used for subscripting.
46. Failed to find a right parenthesis for subscripting.
49. Failed to find a valid formula following an equal sign $\langle = \rangle$ in an assignment statement.
62. Failed to find a valid formula following an IF or WHILE statement.
64. Following what appeared to be a valid (edit mode) program statement, additional characters were found.
82. Missing the ENDP statement.
84. Statement following ENDP statement.
93. Failed to find a THEN beginning the logical line following a logical line beginning with an IF statement.
98. Failed to find an ENDI after a THEN or ELSE statement.
101. Failed to find a logical line beginning with a DO following a logical line beginning with a WHILE statement.
103. Failed to find an ENDW statement after a DO statement.
128. A left "did not have a matching right".
135. Unrecognized character.
146. Unrecognized character following a "?" for KEYB input.
147. Unrecognized character following a numeric constant.
157. A negative sign must be followed by a digit or a decimal point.
158. A decimal point which is not preceded by a digit must be followed by a digit.
168. An E in E-notation must be followed by a negative sign or a digit.
175. Unrecognized character without a preceding "?" encountered for KEYB input.

Capacity (Storage overflow)

1. Stack underflow or overflow.
2. Maximum number of constants exceeded (60).
3. Buffer or stack has overflowed. Statement is too complex.
4. Number is too large or too small (exceeds the permissible range of $\pm 3.618502E\pm 75$).
5. The exponent is too large for an E-notation number.
6. Program is over 255 bytes long.
142. Out of stack space.
156. Out of data space at a request for program or I/O buffer storage.
206. Out of stack space.
220. Out of data space at a request for array, symbol table, or constant table storage. Even if there is significant unused memory, this error message may appear because there is insufficient space for intermediate results (e.g., the concatenation of two arrays where the concatenated result exceeds the available space).

Appendix D: Topic Index

This index includes KEYWORDS, COMMANDS, and all the general topics included in the index, plus the words necessary to understand the error statements.

- ABSolute value, 99
- AND, 100
- ArcCos, Arcsin, Arctan, 53
- argument, 87–89, 141
- ARRAY, 37, 101
- array, see also row variable, 37–41, 141
- assignment, 17–18, 75, 141
- ⟨BACK⟩, 2, 14–15, 17, 71
- BARColor, 5, 77
- BARHeight, 4–5, 11, 78
- block, 23
- bounds, 141
- capacity error, 62
- CEILing, 103
- character constant, 18, 79
- colors, chart of, 5, 77
- commands, see also system commands, 71
- concatenation, see join, 35, 96
- conditionals, 24, 141
- COSine, 103
- cursor, 2, 14, 141
- debugging, 61
- deleting a line, 16–17
- displaying a value, 80
- DO, see WHILE
- EDIT, 13
- edit mode, 13–21, 63, 141
- editing keys, 2–3, 71–72
- ELSE, see IF
- empty array, 89, 101
- ENDEdit, 14, 17
- ENDIf, see IF
- ENDProgram, 14, 17
- ENDWhile, see WHILE
- EQuals, 104
- equals sign (=), see variables, assignment
- ERASe, 43
- ⟨ERASE⟩, 2, 16–17, 72
- error messages, 61–68
 - recovery, 62–67
- evaluate, 141
- EXIT, 29–30
- execute a program, 15
- EXPOnent of e 105
- exponent, 94–95

- exponential function, 94–95
- expression, 17
- factorial, 98
- FLOOR, 106
- formula, 141
- functions, 86–123
- GE, 107
- GT, 108
- grads, conversion to degrees, radians, 104
- IF, 23–25, 81
- immediate mode, 3, 13–21, 62, 141
- INDEX, 38, 109
- infinite loops, 67
- input, also see KEYBoard, 11, 18, 82, 141
- insert a line, 16–17
- inserting the cartridge, 2
- inverse trig functions, 53
- join function, 35, 96
- KEYBoard, 18, 82, 141
- keyword, 21, 145
- LE, 111
- line, logical, 5–6, 142
 - visual, 5–6
- LIST, 73
- LOAD, 45, 47, 73
- LOG, 112
- logarithmic plotting, 78
- logical line, 5–6
- lower case, 2, 14, 142
- LT, 111
- ⟨MASTER CONTROL⟩, 3, 15
- mathematical functions, table of, 138
- MAXimum, 36, 39, 113
- memory, 43, 143
- MINimum 36, 39, 115
- minus sign, 10
- mixed functions, 87
- MODulo, 116
- negative sign, 3, 10
- NE, 117
- nested structures, 30
- ⟨NEXT⟩, 2, 14–15, 71
 - to restart program, 69
- NOT, 118
- OFFTrace, 69, 83
- ONTRace, 69, 83
- OR, 119
- output, 11, 80
- pi, 104

- pointer, 34, 142
- precedence, 8
- ⟨PREVIOUS⟩, 3, 16, 17, 71
- PROGram, 13
- program name, 84
- prompt, 18–19, 142
- RANDom, 120
- ravel, 96
- reduction function, 33, 97
- replace a line, 16–17
- reserved words, 16, 145
- restart a program, 69
- row numbers, 4
- row variable, 9–10, 31–41, 142
 - using functions with, 32
- running a program, 15
- sample programs, 49
- SAVE, 43, 47, 74
- scalar, 142
- ⟨SHIFT⟩, 2
- showing a value, 80
- SIGN, 121
- SINe, 121
- SIZE, 35, 122
- ⟨SPACE⟩, 2
- ⟨SPECIAL⟩, 3, 17, 71
- statement, 75, 142
- ⟨STOP⟩, 67, 69
- subroutines, 49, 84
- subscript, or pointer, 34, 142
- syntax, 3
- syntax error, 62
- system commands, 21, 71
- TANgent, 123
- tape, saving data on, 43–47
 - errors, 44, 46, 62
- THEN, see IF
- times (x), 3, 15
- trigonometric functions, 53
- upper case, 3, 14, 142
- utility programs, 49–52
- value, 17, 131
- value error, 61, 132
- variables, 6–7, 21, 142
 - legal names, 7–8
 - assignment, 6–7
- VERIfy, 46, 74
- WHILe, 27, 85

Appendix D: Topic Index

workspace, 43, 142

+, 90

-, 91

x, 92

÷, 93

* see exponent

, see join function, ravel

∅ see negative sign

π see pi

/ see reduction function

'' see character constant

\$ see variables, legal names

Appendix E: Application Program Index

ADD, counts, 17
ARCCosine, computes arccosine, 54
ARCSine, computes arcsine, 54
ARCTangent, computes arctangent, 53
AVERage, computes average, 40
BALLgame, measures score, 29
BARS, barcolor as function of input number, 25
BARS, plays with bar graphs, 84
CHAIN, chained addition and subtraction, 83
COLR, names colors, 55
Combinations, 99
Compound conditions, 101
CONDitional, tests positive/negative numbers, 24
COUNT, counts to ten, 63
CTOJ, expresses dates as integers, 105
DCF, discounts cashflows, 110
DEC, decision analysis, 114
DECImal, finds decimal part of number, 51
DICE, rolls dice, 121
ERR, illustrates value errors, 65–66
family budgeting, 102
FMAX, finds subscript of largest value in array, 56
FS, generates Fibonacci series, 86
GEN, finds prime numbers, 116
geometric mean, 93
GNUM, number guessing, 59
GOOFy, counts, 28
LAST, finds equation root, 30
LOGP, logarithmic plotting, 78
Ir circuit, 106
mean absolute deviation, 99
MILLion, illustrates value error, 66
MORE, illustrates conceptual error, 28
mortgage payments, 95
net present value, 96
parallel resistance, 94
PAUSE, slows other programs down, 52
payment on a loan, 113
polar coordinates, 104
polynomial evaluation, 95
profit or loss, 109
PVROund, rounds to place value, 52
QUAD, computes quadratic roots, 82
random values, 120
REV, reverses values in an array, 84
RISE, generates rising bar, 23
ROUND, rounds number, 51

- rounding, 103
- SAFX, checks bounds for given value, 57
- SAMP, illustrates list search, 119
- SCALE, scales barheights, 52
- SHIFt, shifts barheights, 55
- SHIFt, shifts values in array, 117
- SHRPshooter, guess number, 58
- significant digits, 107
- SINE, traces sine function, 57
- STAR, loop prints “twinkle, twinkle”, 67
- STAT, analyses data, 79
- TEST, illustrates value error, 64
- TRYDeci, tests program DECI, 51
- TURN, rotates bars, 144
- TWOS, two times table, 13
- VECA, vector summation, 122

Appendix F: Glossary of Terms

argument	a value used by a function or procedure. In the statement “ARRAY 11”, the number 11 is the argument for ARRAY.
array	a row variable. A variable that has in it many values. JETS = 3 17 7 6 is an array named JETS with 4 values it in. See also pointer , subscript , and the keywords ARRAY and INDEX.
assign	give (a value to a variable). In the statement TOP = 63, the value 63 is assigned to the variable TOP. The assignment symbol is =.
bounds	the limits of a row variable. In APL/S, the lower bound is always 1. The upper bound is the number of values in the row variable. If you have a row variable JUMP, and JUMP = 3 22 5 7 109 4 the lower bound is 1, and the upper bound is 6, since JUMP has six values. The statement JUMP (8) would be an error, since 8 is out of bounds – it is larger than the number of values JUMP has. See pointer below.
condition	a relationship between two quantities, which is either true or false. If the condition is true, its value is 1; if the condition is false, its value is 0. There are 6 conditions: EQ (Equals), NE (Not Equal to), LT (Less Than), GT (Greater Than), LE (Less than or Equal to), and GE (Greater than or Equal to). Some examples are (TIME LT 6) or (HITE EQ AGE).
cursor	the tiny square on the tv screen that shows where letters typed on the keyboard will appear. See also immediate , mode , edit mode , upper case , and lower case .
edit mode	the mode the computer has to be in to create or change a program. Edit mode is shown when the cursor is black, and the U or L inside it is white.
evaluate	find the numerical value of a statement or formula or condition. When the computer executes a line like COST = 5 + SHIP, it evaluates “5 + SHIP” before it assigns the value to cost.
formula	a set of relationships between numbers. The following are formulae: 37+2; BOAT – PROW; COS 0.5.
immediate mode	a mode in which the computer will immediately execute legal commands, including programs. Immediate mode is shown by a white cursor with either a black U or L inside it.
input	values which are given to a program by the user. In APL/S, input is done with the KEYBoard command.
keyboard	one of the words or symbols which make up the language APL/S. Each keyword tells VideoBrain to do something unique.

logical line	the characters between <NEXT> and <NEXT> put in in edit mode. This is not the same as the visual line.
lower case	the condition in which the computer takes in the symbols printed on the lower half of a key. In order to get lower case, the <SHIFT> key is pressed. When VideoBrain is in lower case, the cursor shows an "L" inside the cursor. See also upper case .
pointer	a number in parentheses behind a row variable which points to a single value of the row variable. In the program lines below, HITS (2) points to the second value in HITS. <pre>HITS = 3 25 417 68 HITS (2)</pre> The line "HITS (2)" will produce the number 25, since that is the second value in HITS.
prompt	the line of a program which asks the user to type in a number. In the lines: <pre>"TIMES WHAT NUMBER ?" MULT = KEYBoard</pre> The line in quotations, "TIMES WHAT NUMBER ?", is the prompt. The variable MULT gets the value which is input from the keyboard. See also input above.
row variable	a variable which has more than one value. JUMP _s =2 3 4 is a row variable. Also called an array.
scalar	a variable which contains only one value. MOM=100, MOM is a scalar.
statement	a logical line. The characters between <NEXT> and <NEXT>. A legal statement is a combination of keywords, constants, and variables which are in an order VideoBrain can understand.
subscript	a pointer. See the definition of pointer above.
upper case	a condition in which the computer takes in the symbols printed on the upper part of the keys. The <SHIFT> key puts you in upper case. When the computer is in upper case, the cursor has a "U" inside it. See also lower case .
variable	a name made up by the user which can be given any number as its value. JUMP, NUTS, MAMA, and JUNE are all variables in APL/S.
workspace	the active memory of the computer. The workspace holds all current values of variables, the names of the variables, and also holds all the current programs. The workspace of VideoBrain in APL/S is 920 bytes. To see how much workspace is left, use the LIST command. The number which appears immediately after you say "LIST" is the number of bytes of empty memory which are left for you to use.

Appendix G: **Storage Management**

APL/S provides 920 bytes of RAM for your programs, variables, and values assigned to variables. After the first four program or variable names you use, each additional name requires four bytes. An additional four bytes are used to either store the value of the variable if it is a scalar, or to signify that the variable is an array. Each element of an array requires four bytes. In programs, each character (e.g., "+"), word (e.g., "THEN") or variable name requires one byte. The exceptions are "IF", "ELSE", "WHILE" and "ENDWHILE" which each use two bytes. Each statement requires one byte.

When you run a program with arrays in it, the program will take more memory to store the values in the array. For example, the program statement "A = ARRA 7" will require four bytes in the program listing and will take an additional 28 bytes to store seven values when the program runs.

APL/S allocates storage dynamically so that you can input any balance of variable names, programs, and data (values) that you wish. You can re-allocate the space by erasing programs and data (values) and then writing in new programs, data and variables. You erase values by erasing the variable name to which they are assigned. Variable names can be erased and replaced with new names but once space has been allocated to variable names, it can only be used for variable names.

When you type the command "LIST", APL/S will first return the number of bytes still available to you. The number includes any space you have freed up by erasing programs or values. It does not include space you have freed for new variable names by erasing old variable names.

Good storage management will not only give you more free space to work with, it will help you organize your work in the clearest possible manner. Some guidelines for good storage management are as follows:

- 1. Erase programs, data, and variable names which you no longer need. Be careful not to erase variables which are used in programs you are keeping.
- 2. Store programs on tape singly and in useful combinations.
- 3. Condense programs where applicable. In general, program readability is more important than brevity. This means you should use short statements and many character constants. If you need to conserve space or if you're writing a utility program to fit in a workspace with other programs, you may want to condense your program by deleting character constants and shrinking several statements into one. For example the statements on the left can be condensed into the single statement on the right below:

Clarity	Brevity
IF y EQ 10 THEN x = 5 ELSE x = 2 ENDIF	x = y EQ 10 x 3 + 2
20 bytes	8 bytes

4. Reduce the number of variables by using expressions instead of naming new variables. For example, the following program to rotate bars:

```

PROG TURN
C = INDEX 7
B = C x 10
WHILE C(1) LE 100
DO 100 - C(1)
    BARH((B=B+10) MOD 130-65)
    BARC C = C+1
ENDW
ENDP

```

120 bytes

```

PROG TURN
C = INDEX 7
WHILE C(1) LE 100
DO 100 - C(1)
    BARH(C x 10 MOD 130-65)
    BARC C = C+1
ENDW
ENDP

```

100 bytes

You can save even more space by using the same variable names in several programs. This invites trouble, however, and should be avoided if at all possible.

5. Assign values to variables in immediate mode and then use the variables in your programs — rather than assigning the values within the program. When you assign values in immediate mode, they are only stored once in conjunction with the variable. When you assign values within the program, they are stored twice — once as part of the program and once again in association with the variable when the program assigns the value to the variable. For example:

```

A = 10 20 30
PROG ABC
A
ENDP

```

20 bytes

```

PROG ABC
A = 10 20 30
A
ENDP

```

32 bytes

6. If you use the same number (eg. 1) several times in your programs, assign the value to a variable in immediate mode (eg. W=1) and then use the variable name in the program instead of the number. Using the variable name takes only one byte each time — the number itself would take four bytes each time.

In the immediate mode, variable names and the values assigned to them continue to take up space until you erase them. The space taken up by expressions you type in is freed up as soon as the expression is evaluated.

Appendix H: APL/S Reserved Words

The following list of words have special meanings in APL/S. These words cannot be used to name variables or programs, and therefore they are called reserved words.

ABS	LIST
AND	LOAD
ARRAY	LOG
BARColor	LT
BARHeight	MAXimum
CEILing	MINimum
COSine	MODulo
DO	NE
EDIT	NOT
ELSE	OFFTTrace
ENDEdit	ONTRace
ENDIf	OR
ENDProgram	PROGram
ENDWhile	RANDom
EQuals	SAVE
ERASe	SIGN
EXIT	SINe
EXP	TANGent
FLOOR	THEN
GE	VERIfy
GT	WHILe
IF	
INDEX	
KEYBoard	
LE	

APL/S is just one of the many exciting VideoBrain cartridges brought to you by the VideoBrain Computer Company. We suggest you try all the VideoBrain Cartridges to help you around the home, educate your children and entertain the whole family.

Money Management

VB-81 Financier™
VB-1000 Money Minder

Communications

CM01 Timeshare

Education

ED01 Music Teacher 1
ED02 Math Tutor 1
ED03 Wordwise™ 1
ED04 Wordwise™ 2
ED05 VideoArtist™
ED06 Lemonade Stand—A Business Simulation
ED07 Musicianship 1
ED09 Historical Simulation—France in the Old Regime

Entertainment

EN01 Gladiator
EN02 Pinball
EN03 Tennis
EN04 Checkers
EN05 Blackjack
EN06 Vice Versa™

Limited 90-Day Warranty on Cartridges:

For 90 days from the date of purchase, VideoBrain Computer Company will repair any defect in material or workmanship in this Cartridge free of charge.

To obtain warranty service, return the Cartridge postpaid, with sales receipt showing date of purchase, to the VideoBrain Service Center with address shown below.

Under no circumstances will VideoBrain Computer Company be liable for any special, incidental, or consequential damages resulting from use or possession of the VideoBrain or its accessories. However, some states do not allow the exclusion or limitation of incidental or consequential damages, so that the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

VideoBrain Computer Company
2950 Patrick Henry Drive
Santa Clara, Ca. 95050
© 1978

Printed in U.S.A.

VideoBrain Computer Co.

An Umtech Company

2950 Patrick Henry Drive
Santa Clara, California 95050
USA
408/988-3020

741 0026